

PATENT COOPERATION TREATY

PCT

NOTIFICATION OF ELECTION

(PCT Rule 61.2)

From the INTERNATIONAL BUREAU

To:

Commissioner
US Department of Commerce
United States Patent and Trademark
Office, PCT
2011 South Clark Place Room
CP2/5C24
Arlington, VA 22202
ETATS-UNIS D'AMERIQUE
in its capacity as elected Office

Date of mailing (day/month/year) 08 December 2000 (08.12.00)	
International application No. PCT/GB00/01691	Applicant's or agent's file reference JSR.P51122PC
International filing date (day/month/year) 03 May 2000 (03.05.00)	Priority date (day/month/year) 04 May 1999 (04.05.99)
Applicant ROBERTS, Derek, Edward et al	

1. The designated Office is hereby notified of its election made:

☒

in the demand filed with the International Preliminary Examining Authority on:

15 November 2000 (15.11.00)

☐

in a notice effecting later election filed with the International Bureau on:

2. The election ☒ was

☐

was not

made before the expiration of 19 months from the priority date or, where Rule 32 applies, within the time limit under Rule 32.2(b).

The International Bureau of WIPO 34, chemin des Colombettes 1211 Geneva 20, Switzerland Facsimile No.: (41-22) 740.14.35	Authorized officer <p style="text-align: center;">Juan Cruz</p> Telephone No.: (41-22) 338.83.38
--	--

This Page Blank (uspto)

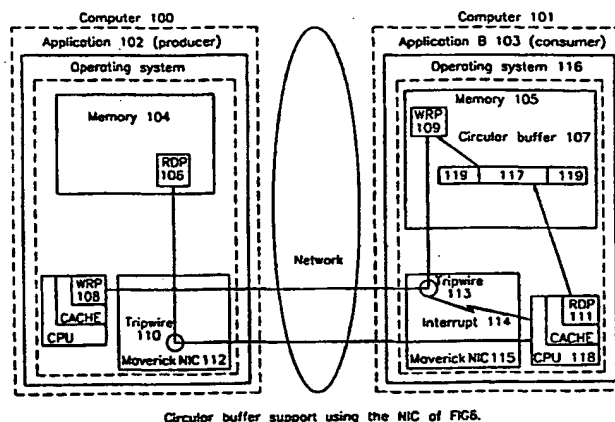


INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁷ : G06F 13/00	A2	(11) International Publication Number: WO 00/67131
		(43) International Publication Date: 9 November 2000 (09.11.00)

(21) International Application Number: **PCT/GB00/01691**(22) International Filing Date: **3 May 2000 (03.05.00)**(30) Priority Data:
9910280.8 **4 May 1999 (04.05.99)** **GB**(71) Applicant (for all designated States except US): **AT & T
LABORATORIES-CAMBRIDGE LIMITED [GB/GB];
24A Trumpington Street, Cambridge CB2 1QA (GB).**

(72) Inventors; and

(75) Inventors/Applicants (for US only): **ROBERTS, Derek,
Edward [GB/GB]; 25 Metcalf Road, Cambridge CB4
2DB (GB). POPE, Steven, Leslie [GB/GB]; 25 Greville
Road, Cambridge CB1 3QJ (GB). MAPP, Glenford, Ezra
[GB/GB]; 30 Mowbray Road, Cambridge CB1 7SY (GB).
HODGES, Stephen, John [GB/GB]; 12 Petersham Mews,
Kensington, London SW7 5NR (GB).**(74) Agent: **ROBINSON, John, Stuart; Marks & Clerk, 4220 Nash
Court, Oxford Business Park South, Oxford OX4 2RU (GB).**(81) Designated States: **US, European patent (AT, BE, CH, CY,
DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT,
SE).****Published***Without international search report and to be republished
upon receipt of that report.*(54) Title: **DATA TRANSFER, SYNCHRONISING APPLICATIONS, AND LOW LATENCY NETWORKS**

(57) Abstract

A synchronous network interface and method of synchronisation between two applications on different computers is provided. The network interface contains snooping hardware which can be programmed to contain triggering values comprising either addresses, address ranges or other data which are to be matched. These data are termed "trip wires". Once programmed, the interface monitors the data stream, including address data, passing through the interface for addresses and data which match the trip wires which have been set. On a match, the snooping hardware can generate interrupts, increment event counters, or perform some other application-specified action. This snooping hardware is preferably based upon Content-Addressable Memory. The invention thus provides in-band synchronisation by using synchronisation primitives which are programmable by user level applications, while still delivering high bandwidth and low latency. The programming of the synchronisation primitives can be made by the sending and receiving applications independently of each other and no synchronisation information is required to traverse the network.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakhstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

DATA TRANSFER, SYNCHRONISING APPLICATIONS, AND LOW LATENCY NETWORKS

This invention, in its various aspects, relates to the field of asynchronous networking, and specifically to: a memory mapped network interface; a method of synchronising between a sending application, running on a first computer, and a receiving application, running on a second computer, the computers each having a memory mapped network interface; a communication protocol; and a computer network. This invention also relates to data transfer and to synchronising applications.

Due to a number of reasons, traditional networks, such as Gigabit Ethernet, ATM, etc., have not been able to deliver high bandwidth and low latency to applications that require them. A traditional network is shown in Fig. 1. To move data from computer 200 to another computer 201 over a network, the Central Processing Unit (CPU) 202 writes data from memory 204 through its system controller 206 to its Network Interface Card (NIC) 210. Alternatively, data may be transferred to the NIC 210 using Direct Memory Access (DMA) hardware 212 or 214. The NIC 210 takes the data and forms network packets 216, which contain enough information to allow them to be routed across the network 218 to computer system 201.

When a network packet arrives at the NIC 211, it must be demultiplexed to determine where the data needs to be placed. In traditional networks this must be done by the operating system. The incoming packet therefore generates an interrupt 207, which causes software, a device driver in operating system 209, to run. The device driver examines the header information of each incoming network packet 216 and determines the correct location in memory 205, for data contained within the network packet. The data is transferred into memory using the CPU 203 or DMA hardware (not shown). The driver may then request that operating system 209 reschedule any application process that is blocked waiting for this data to arrive. Thus there is a direct sequence from the arrival of incoming packets to the scheduling of the receiving application. These

networks therefore provide implicit synchronisation between sending and receiving applications and are called synchronous networks.

It is difficult to achieve optimum performance using modern synchronous network hardware. One reason is that the number of interrupts that have to be processed increases as packets are transmitted at a higher rate. Each interrupt requires that the operating system is invoked and software is executed for each packet. Such overheads both increase latency and the data transfer size threshold at which the maximum network bandwidth is achieved.

These observations have led to the development of asynchronous networks. In asynchronous networks, the final memory location within the receiving computer for received data can be computed by the receiving NIC from the header information of a received network packet. This computation can be done without the aid of the operating system.

Hence, in asynchronous networks there is no need to generate a system interrupt on the arrival of incoming data packets. Asynchronous networks therefore have the potential of delivering high bandwidth and low latency; much greater than synchronous networks. The Virtual Interface Architecture (VIA) is emerging as a standard for asynchronous networking.

Memory-mapped networks are one example of asynchronous networks. An early computer network using memory mapping is described in US patent No. 4,393,443.

A memory-mapped network is shown in Fig. 2. Application 222 running on Computer 220 would like to communicate with application 223 running on Computer 221 using network 224. A portion of the application 222's memory address space is mapped using the computer 220's virtual memory system onto a memory aperture of the NIC 226 as shown by the application's page-tables 228 (these page-tables and their use is well known in the art). Likewise, a portion of application 223's memory address space is

mapped using computer 221's virtual memory system onto a memory aperture of the NIC 229 using the application 223's page-tables 231. Software is usually required to create these mappings, but once they have been made, data transfer to and from a remote machine can be achieved using a CPU read or write instruction to a mapped virtual memory address.

If application 222 were to issue a number of processor write instructions to this part of its address space, the virtual memory and I/O controllers of computer 220 will ensure that these write instructions are captured by the memory aperture of the NIC 226. NIC 226, determines the address of the destination computer 221 and the address of the remote memory aperture 225 within that computer. Some combination of this address information can be regarded as the network address, which is the target of the write.

All the aperture mappings and network address translations are calculated at the time that the connection between the address spaces of computers 220 and 221 is made. The process of address lookups and translations at each stage in the system can be carried out using hardware.

After receiving a write, NIC 226 creates network packets using its packetisation engine 230. These packets are forwarded to the destination computer 221. At the destination, the memory aperture addresses of the incoming packets are remapped by the packet handler onto physical memory locations 227. The destination NIC 229 then writes the incoming data to these physical memory locations 227. This physical memory has also been mapped at connection set-up time into the address space of application 223. Hence application 223 is able, using page-tables 231 and the virtual memory system, to access the data using processor read and write operations.

Commercial equipment for building memory-mapped networks is available from a number of vendors, including Dolphin Interconnect Solutions. Industry standards, such as Scalable Coherent Interface (SCI) (IEEE Standard 1596-1992), have been defined for

building memory mapped networks, and implementations to the standards are currently available. -

SCI is an example of an asynchronous network standard, which provides poor facilities for synchronisation at the time of data reception. A network using SCI is disclosed in US Patent No. 5,819,075. Figure 3 shows an example of an SCI- like network, where application 242 on computer 240 would like to communicate with application 243 on computer 241. Let us suppose that application 243 has blocked waiting for the data. Application 242 transmits data using the methods described above. After sending the data, application 242 must then construct a synchronisation packet in local memory, and program the event generator 244, in NIC 246, to send the synchronisation packet 248, to the destination node.

On receiving synchronisation packet 248, the NIC 245 on computer 241, invokes its event handler 247, which generates an interrupt 249 allowing the operating system 248 to determine that application 243 is blocked and should be woken up. This is called out-of-band synchronisation since the synchronisation packet must be treated as a separate and distinct entity and not as part of the data stream. Out-of-band synchronisation greatly reduces the potential of memory-mapped networks to provide high bandwidth and low latency.

In other existing asynchronous networks, such as the newly emerging Virtual Interface Architecture (VIA) standard and the forthcoming Next Generation Input/Output (NGIO) standard, some support is provided for synchronisation. A NIC will raise a hardware interrupt when some data has arrived. However, the interrupt does not identify the recipient of the data, instead only indicates that some data has arrived for some communicating end-point.

While delivery of data can be achieved solely by hardware, the software task of scheduling between a large number of applications, each handling received data, becomes difficult to achieve. Software, known as a device driver, is required to examine

a large number of memory locations to determine which applications have received data. It must then notify such applications that data has been delivered to them. This might include a reschedule request to the operating system for the relevant applications.

Other known data transfer techniques are disclosed in EP 0 600 683, EP 0 359 137, EP 0 029 800, US 5 768 259, US 5 550 808 and JP 600211559.

The present invention, in its various aspects, is defined in more detail in the appended claims to which reference should now be made.

A first aspect of the invention provides a method of synchronising between a sending application on a first computer and a receiving application on a second computer, each computer having a main memory, and at least one of the computers having an asynchronous network interface, comprising the steps of:

- providing the asynchronous network interface with a set of rules for directing incoming data to memory locations in the main memory of the second computer;

- storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the applications;

- receiving, at the network interface, a data stream being transferred between the applications;

- comparing at least part of the data stream received with the stored triggering values;

- if the compared part of the data stream matches any stored triggering value, indicating that the triggering value has been matched; and

- storing the data received in the main memory of the second computer at one or more memory location(s) in accordance with the said rules.

Another aspect of the invention provides an asynchronous network interface for use in a host computer having a main memory and connected to a network, the interface comprising:

means for storing a set of rules for directing incoming data to memory locations in the main memory of the host computer;

a memory for storing one or more triggering value(s), each value representing a state of a data transfer between two or more applications in the computer network;

a receiver for receiving a data stream being transferred between two or more applications in the computer network; comparison means for comparing at least part of the data stream received by the network interface with the stored triggering values; and

a memory for storing information identifying any matched triggering values.

A further aspect of the invention provides a method of passing data between an application on a first computer and remote hardware within a second computer or on a passive backplane, the first computer having a main memory and an asynchronous network interface, the method comprising the steps of:

providing the asynchronous network interface with a set of rules for directing incoming data to memory or I/O location(s) of the remote hardware;

storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the application and the hardware;

receiving, at the network interface, a data stream being transferred between the application and the hardware;

comparing at least part of the data stream received with the stored triggering value(s);

indicating that a triggering value has been matched, if any compared part of the data stream matches a triggering value;

storing data transmitted in memory or I/O location(s) of the remote hardware in accordance with the said rules; and

storing the data received in the main memory of the computer at one or more memory location(s) in accordance with the said rules.

A further aspect of the invention provides a method of arranging data transfers from one or more applications on a computer, the computer having a main memory, an

asynchronous network interface, and a Direct Memory Access (DMA) engine having a request queue address common to all the applications, comprising the steps of:

the application requesting the network interface to store a triggering value corresponding to a property of the data block to be transferred;

an application requesting the DMA engine to transfer a block of data;

the network interface storing a triggering value corresponding to a property of the data block to be transferred, along with an identification of the application which requested the DMA transfer;

the network interface monitoring the data stream being sent by the applications and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

A yet further aspect of the invention provides a method of transferring data from a sending application on a first computer to a receiving application on a second computer, each computer having a main memory, and a memory mapped network interface, the method comprising the steps of:

creating a buffer in the main memory of the second computer for storing data being transferred as well as data identifying one or more pointer memory location(s);

storing at said pointer memory location(s) at least one write pointer and at least one read pointer for indicating those areas of the buffer available for writes and for reads;

in dependence on the values of the WRP(s) and RDP(s), the sender application writing to the buffer;

updating the value of the WDP(s), after a write has taken place, to update the indication of the areas of the buffer available for reads and writes;

in dependence on the values of WRP(s) and RDP(s), the receiver application reading from the buffer; and

updating the value of the RDP(s), after a read has taken place, to update the indication of the areas of the buffer available for reads and writes.

Another aspect of the invention provides a computer network comprising two computers, the first computer running a sending application and the second computer running a receiving application, each computer having a main memory and a memory mapped network interface, the main memory of the second computer having: a buffer for storing data being transferred between computers as well as data identifying one or more pointer memory location(s);

means for reading at least one write pointer (WRP) and at least one read pointer (RDP) stored at (a) pointer memory location(s), for indicating those areas of the buffer available for writes and those areas available for reads;

the network interface of the second computer comprising:

a memory mapping;

means for reading data from the buffer in accordance with the contents of the WRP(s) and RDP(s); and

means for updating the value of the RDP(s), after a read has taken place, to update the indication of the areas of the buffer available for reads and writes.

A further aspect of the invention provides a method of sending a request from a client application on a first computer to a server application on a second computer, and sending a response from the server application to the client application, both computers having a main memory and a memory mapped network interface, the method comprising the steps of:

(A) providing a buffer in the main memory of each computer;

(B) the client application, providing software stubs which produce a marshalled stream of data representing the request;

(C) the client application sending the marshalled stream of data to the server's buffer;

(D) the server application unmarshalling the stream of data by providing software stubs which convert the marshalled stream of data into a representation of the request in the server's main memory;

(E) the server application processing the request and generating a response;

(F) the server application providing software stubs which produce a marshalled stream of data representing the response;

(G) the server application sending the marshalled stream of data to the client's buffer; and

(H) the client application unmarshalling the received stream of data by providing software stubs which convert the received marshalled stream of data into a representation of the response in the client's main memory.

Another aspect of the invention provides a method of arranging data for transfer as a data burst over a computer network comprising the steps of: providing a header comprising the destination address of a certain data word in the data burst, and a signal at the beginning or end of the data burst for indicating the start or end of the burst, the destination addresses of other words in the data burst being inferrable from the address in the header.

A further aspect of the invention provides a method of processing a data burst received over a computer network comprising the steps of:

reading a reference address from the header of the data burst, and

calculating the addresses of each data word in the burst from the position of that data word in the burst in relation to the position of the data word to which the address in the header corresponds, and from the reference address read from the header.

Another aspect of the invention provides a method of interrupting transfer of a data burst over a computer network comprising the steps of:

halting transfer of a portion of the data burst which has not yet been transferred, thereby splitting the data burst into two burst sections, one which is transferred, and one waiting to be transferred.

A further aspect of the invention provides a method of restarting the transfer of a data burst, after the transfer of that data burst has been interrupted, the method comprising the steps of:

calculating a new reference address for the untransferred data burst section from the address contained in the header of the whole data burst, and from the position in the whole data burst of the first data word of the untransferred data burst section in relation to the position of the data word to which the address in the header corresponds;

providing a new header for the untransferred data burst section comprising the new reference address; and

transmitting the new header along with the untransferred data burst section.

The first aspect of the present invention addresses the synchronisation problem for memory mapped network interfaces. The present invention uses a network interface, containing snooping hardware which can be programmed to contain triggering values comprising either addresses, address ranges, or other data which are to be matched. These data are termed 'Tripwires'. Once programmed, the interface monitors the data stream, including address data, passing through the interface for addresses and data which match the Tripwires which have been set. On a match, the snooping hardware can generate interrupts or increment event counters, or perform some other application specified action. This snooping hardware is preferably based upon Content Addressable Memory (CAM). References herein to the "data stream" refer to the stream of data words being transferred and to the address data accompanying them.

The invention thus provides in-band synchronisation by using synchronisation primitives which are programmable by user level applications, while still delivering high bandwidth and low latency. The programming of the synchronisation primitives can be made by the sending and receiving applications independently of each other and no synchronisation information is required to traverse the network.

A number of different interfaces between the network interface and an application can be supported. These interfaces include VIA and the forthcoming Next Generation Input/Output (NGIO) standard. An interface can be chosen to best match an application's requirements, and changed as its requirements change. The network

interface of the present invention can support a number of such interfaces simultaneously.

The Tripwire facility supports the monitoring of outgoing as well as incoming data streams. These Tripwires can be used to inform a sending application that its DMA send operations have completed or are about to complete.

Memory-Mapped network interfaces also have the potential to be used for communication between hardware entities. This is because memory mapped network interfaces are able to pass arbitrary memory bus cycles over the network. As shown in Fig. 4, it is possible to set up a memory aperture 254, in the NIC 252 of Computer 250, which is directly mapped via NIC 259, onto an address region 257 of the I/O bus 253 of passive backplane 251.

Using existing memory mapped interfaces, such as DEC Memory Channel or Dolphin SCI, an application running on Computer 250, which requires use of the hardware device 255, would require a (usually software) process to interface between itself and the Network Interface card (NIC) 252. This is because the NIC 252, would not appear at the hardware level in computer 250 as an instance of the remote hardware device 255, but instead as a network card which has a memory aperture 254 mapped onto the hardware device.

In a further aspect of the invention, we have appreciated that the interface of the present invention can be programmed to present the same hardware interface as the remote hardware device 255, and so appear at the hardware level in computer 250 to be an instance of the remote hardware device. If the network card 252 were an interface according to the present invention, so programmed, the remote hardware device 255 would appear as physically located within computer 250, in a manner transparent to all software. The hardware device 255, is able to be physically located both at the remote end of a dedicated link, or over a general network. The invention will support both

general networking activity and remote hardware communication simultaneously on a single network card.

Another aspect of the invention relates to a link-level communication protocol which can be used to support cut-through routing and forwarding. There is no need for an entire packet to arrive at a NIC, or any other network entity supporting the communication protocol, before data transmission can be started on an outgoing link. The invention also allows large bursts of data to be handled effectively without the need for a small physical network packet size such as that employed by an ATM network, it being possible to dynamically stop and restart a burst and regenerate all address information using hardware.

A preferred embodiment of the various aspects of the invention will now be described with reference to the drawings in which:

Figure 5 shows two or more computers connected by an embodiment of the present invention, using Network Interface Cards (NICs);

Figure 6 shows in detail the various functional blocks comprising the NICs of Figure 5;

Figure 7 shows the functional blocks of the NIC employed within a Field Programmable Gate Array (FPGA);

Figures 8 and 8e shows the communication protocol used in one embodiment of the invention;

Figure 9 shows schematically hardware communication according to an embodiment of the invention;

Figure 10 shows schematically a circular buffer abstraction according to one embodiment of the invention;

Figure 11 shows schematically the system support for discrete message communication using circular buffers;

Figure 12 shows a client-server interaction according to an embodiment of the invention;

Figure 13 shows how the system of the present invention can support VIA;

Figure 14 shows outgoing stream synchronisation according to an embodiment of the present invention;

Figure 15 shows a client-server interaction according to an embodiment of the invention using a hardware data source;

Figure 16 shows an apparatus for synchronising an end-point application and constituting an embodiment of the invention;

Figure 17 shows another apparatus for synchronising an end-point application and constituting an embodiment of the invention;

Figures 18 to 23 show examples of actions which may be performed by the apparatuses of Figures 16 and 17;

Figure 24 illustrates the format of a data burst with implied addresses;

Figure 25 illustrates an interruption in forwarding a burst of the type shown in Figure 24;

Figure 26 illustrates forwarding of the rest of the burst;

Figure 27 illustrates coalescing of two data bursts;

Figure 28 illustrates "transparent" communication over a network between an application running on a computer and remote hardware; and

Figure 29 illustrates applications of various tripwires at different locations in a computer.

Referring to Figure 5, computers 1, 2 use the present invention to exchange data. A plurality of other computers such as 3, may participate in the data exchange if connected via optional network switch 4.

Each computer 1, 2 is composed of a microprocessor central processing unit 5,57, memory 6,60, local cache memory 7,57, and system controller 8,58. The system controller 8,58 interacts with its microprocessor 5,57 to allow the microprocessor to exchange data with devices attached to I/O bus 9. Attached to I/O bus 9,59 are standard peripherals, such as a video adapter 10. Also attached to I/O bus 9,59 is one or more network interfaces, in the form of NICS 11,56 which represent an embodiment of this invention. In computers 1, 2 the I/O bus is a standard PCI bus conforming to PCI Local

Bus Specification, Rev. 2.1, although any other bus capable of supporting bus master operations can be used with suitable modification of System Controller peripherals, such as video card 10, and the interface to NIC 11,56.

Referring to Figure 6, each NIC comprises a memory 18, 19, 20 for storing triggering values, a receiver 15 for receiving a data stream, a comparator for comparing part of the data stream with the triggering values and a memory 23 for storing information which will identify matched triggering values. More specifically, in the preferred embodiment each NIC 56, 11 is composed of a PCI to Local Bus bridge 12, a control Field Programmable Gate Array (FPGA) 13, transmit (Tx) serialiser 14, fibre-optic transceiver 15, receive (Rx) de-serialiser 16, address multiplexer and latch 17, CAM array 18, 19, 20, boot ROMs 21 and 22, static RAM 23, FLASH ROM 24, and clock generator and buffer 25, 26. Figure 6 also shows examples of known chips which could be used for each component, for example boot ROM 21 could be an Altera EPC1 chip.

Referring to Figure 7, FPGA 13 is comprised of functional blocks 27-62. The working of the blocks will be explained by reference to typical data flows.

Operation of NIC 11 begins by computer 1 being started or reset. This operation causes the contents of boot ROM 21 to be loaded into FPGA 13 thereby programming the FPGA and, in turn, causing state machines 28, 37, 40, 43, 45, 46 and 47 to be reset.

Clock generator 25 begins running and provides a stable clock for the Tx serialiser 14. Clock buffer/divider 26 provides suitable clocks for the rest of the system. Serialiser 14 and de-serialiser 16 are reset and remain in a reset condition until communication with another node is established and a satisfactory receive clock is regenerated by de-serialiser 16.

PCI bridge 12 is also reset and loaded with the contents of boot ROM 22. Bridge 12 can convert (and re-convert at the target end) memory access cycles into I/O cycles and support legacy memory apertures, and as the rest of the NIC supports byte-enabled

(byte-wide as well as word-wide) transfers, ROM 22 can be loaded with any PCI configuration space information, and can thus emulate any desired PCI card transparently to microprocessor 5.

Immediately after reset, FLASH control state machine 47 runs and executes a simple microcode sequence stored in FLASH memory 24. Typically this allows the configuration space of another card such as 69 in Figure 9 to be read, and additional information to be programmed into bridge 12. Programming of the FLASH memory is also handled by state machine 47 in conjunction with bridge 12.

Data transfer could in principle commence at this point, but arbiter 40 is barred from granting bus access to Master state machine 37 until a status bit has been set in one of the internal registers 49. This allows software to set up the Tripwires during the initialisation stage.

Writes from computer 1 to computer 2 take place in the following manner. Microprocessor 5 writes one or more words to an address location defined by system controller 8 to lie within NIC 11's address space. PCI to local bus bridge 12 captures these writes and turns them into local bus protocol (discussed elsewhere in this document). If the writes are within the portion of the address space determined to be within the local control aperture of the NIC by register decode 48, then the writes take place locally to the Content Addressable Memory appropriate register, (CAM), Static RAM (SRAM) or FLASH memory area. Otherwise target state machine 28 claims the cycles and forwards them to protocol encoder 29.

At the protocol encoder, byte-enable, parity data and control information are added first to an address and then to each word to be transferred in a burst, with a control bit marking the beginning of the burst and possibly also a control bit marking the end of the burst. The control bit marking the beginning of the burst indicates that address data forming the header of the data burst comprises the first "data" word of the burst. Xon/Xoff-style management bits from block 31 are also added here. This protocol,

specific to the serialiser 14 and de-serialiser 16 is also discussed elsewhere in this document.

Data is fed on from encoder 29 to output multiplexer 30, reducing the pin count for FPGA 13 and matching the bus width provided by serialiser 14. Serialiser 14 converts a 23-bit parallel data stream at 62MHz to a 1-bit data stream at approximately 1.5Gbit/s; this is converted to an optical signal by transceiver 15 and carried over a fibre-optic link to a corresponding transceiver 15 in NIC 56, part of computer 2. It should be noted that other physical layers and protocols are possible and do not limit the scope of the invention.

In NIC 56, the reconstructed digital signal is clock-recovered and de-serialised to 62MHz by block 16. Block 32 expands the recovered 23 bits to 46 bits, reversing the action of block 30. Protocol decoder 33 checks that the incoming words have suitable sequences of control bits. If so, it passes address/data streams into command FIFO 34. If the streams have errors, they are passed into error FIFO 35; master state machine 37 is stopped; and an interrupt is raised on microprocessor 57 by block 53. Software is then used to decipher the incoming stream until a correct sequence is found, whereupon state machine 37 is restarted.

When a stream arrives at the head of FIFO 34, master state machine 37 requests access to local bus 55 from arbiter 40. When granted, it passes first the address, then the following data onto local bus 55. Bridge 12 reacts to this address/data stream by requesting access to I/O bus 59 from system controller 58. When granted, it writes the required data into memory 60.

Reads of computer 2's memory 60 initiated by computer 1 take place in a similar manner. However, state machine 28 after sending the address word sends no other words, rather it waits for return data. Data is returned because master state machine 37 in NIC 56 reacts to the arrival of a read address by requesting a read of memory 60 via I/O bus 59 and corresponding local bus bridge 12. This data is returned as if it were

write data flowing from NIC 56 to NIC 11, but without an initial address. Protocol decoder 33 reacts to this addressless data by routing it to read return FIFO 36, whereupon state machine 28 is released from its wait and the microprocessor 5's read cycle is allowed to complete. Should the address region be marked in NIC 56's bridge 12 as read-prefetchable, then a number of words are returned; if state machine 28 continues requesting data as if from a local bus burst read, then subsequent words are fulfilled directly from read return FIFO 36.

Should NIC 56 need to raise an interrupt on microprocessor 5, remote interrupt generator 54 causes state machine 28 to send a word from NIC 56 to a mailbox register in NIC 11's bridge 12. This will have been configured by software to raise an interrupt on microprocessor 5.

Inevitably, since the clocks 25 in NICs 11 and 56 will run at slightly different frequencies, there will be occasional overrun conditions. Where the command FIFO 34 exceeds a pre-programmed threshold value, an Xoff bit is sent to the corresponding protocol encoder 29. This bit causes the encoder to request that the sending state machine 28 stops, if necessary in mid burst. Logic in bridge 12 takes care of restarting the data burst when the corresponding Xon is received some time later. This logic calculates a new reference address for the unsent part of the data burst, using the reference address in the header of the whole data burst, and from a count of the number of data words which are sent before the transfer is stopped. As, in this embodiment, successive data words in a burst have successively incrementing destination addresses, the destination address of the first data word in the unsent part of the data burst can easily be calculated.

It is also possible that data may be read out of FIFO 34 faster than it is written in. In the event of this happening, master state machine 37 uses pipeline delay 38 to anticipate the draining of FIFO 34 and to terminate the data burst on local bus 55. It then uses the CAM address latch/counter 41 to restart the burst when more data arrives in FIFO 34.

'Tripwires' are triggering values, such as addresses, address ranges or other data, that are programmed into the NIC to be matched. Preferably, the triggering values used as tripwires are addresses. To meet timing requirements during address match cycles (as data flows through the NIC), three CAM devices are pipelined to reduce the match cycle time from around 70 nanoseconds to less than 30 nanoseconds.

The programming of Tripwires takes place by microprocessor 5 writing to PCI bridge 12 via system controller 8 and I/O bus 9. For the purpose of writing the Tripwire data, CAM array 18, 19, 20 appears like conventional RAM to microprocessor 5. For write cycles, this is done by CAM controller 43 generating suitable control signals to enable all three CAMs 18, 19, 20 for write access. Address latch 44 passes data to the CAMs unmodified. Address multiplexer 41 is arranged to pass local bus data out on the CAM address bus where it is latched at the moment addresses are valid on the local bus by latch 17. For read cycles, the process is similar, except that only CAM 18 is arranged to be enabled for read access, and address latch/counter 44 has its data flow direction reversed. So far as microprocessor 5 is concerned, it sees the expected data returned, since the memory arrays in CAMs 18, 19, 20 either contain the same data, or internal flags indicating that particular segments of the memory array have not yet been written and should not participate in match cycles.

Owing to the nature of the address/data bus being comprised of bursts of data, according to the preferred local protocol, the actual data stream cannot be used for monitoring address changes. A burst starts with the address of the first data word followed by an arbitrary number of data words. The address of the data words is implicit and increments from the start address. For normal inbound or outbound data transfer operations, address latch/counter 44 is loaded with the address of each new data burst, and incremented each time a valid data item is presented on internal local bus 55. CAM control state machine 43 is arranged to enable each CAM 18, 19, 20 in sequence for a compare operation as each new address is output by latch/counter 44. This sequential enabling of the CAMs combined with their latching properties permits the access time for a comparison operation to be reduced by a factor of three (there being

three CAMs in this implementation, other implementations being possible) from 70ns to less than 30ns. The CAM op-code for each comparison operation is output from one of the internal registers 49 via address multiplexers 41 and 17. The op-code is actually latched by address multiplexer 17 at the end of a read/write cycle, freeing the CAM address bus to return the index of matched Tripwires after comparison operations.

The Tripwire data (i.e. the addresses to be monitored) is written to sequential addresses in the CAM array. During the comparison operation (cycle), all valid Tripwires are compared in parallel with the address of the current data, be it inbound or outbound. During the operation, masking operations may be performed, depending on the type of CAM used, allowing certain bits of the address to be ignored during the comparison. In this way, a Tripwire may actually represent a range of addresses rather than one particular address.

When the CAM array signals a match found (i.e. a Tripwire has been hit), it returns the address of the Tripwire (its offset in the CAM array) via the CAM address bus to the tripwire FIFO 42. Two courses of action are then possible, depending on how internal registers 49 have been programmed.

One course of action is for state machine 45 to request that an interrupt be generated by management logic 53. In this case, an interrupt is received by microprocessor 5, and software is run which services the interrupt. Normally this would involve microprocessor 5 reading the Tripwire address from FIFO 42, matching the address with a device-driver table, signalling the appropriate process, marking it runnable and rescheduling.

An alternative course of action is for state machine 45 to cause records to be read from SRAM 23 using state machine 46. A record comprises a number of data words; an address and two data words. These words are programmed by the software just before the Tripwire information is stored in the CAM. When a Tripwire match is made, the address in LATCH 44 is left shifted by two to form an address index for SRAM 23.

The first word is then read by state machine 46 and placed on local bus 55 as an address in memory 6. A fetch-and-increment operation is then performed by state machine 45, using the second and third words of the SRAM record to first AND and then OR, or else INCREMENT the data referred to in memory 6. A bit in the first word read by the state machine will indicate which operation it should take. In the case of an INCREMENT, the first data word also indicates the amount to increment by.

These alternatives enable the implementation of such primitives as an event counter incremented on tripwire matches, or the setting of a system reschedule flag. This mechanism enables multiple applications to process data without the requirement for hardware interrupts to be generated after receipt of each network packet.

While in the case of the interrupt followed by a Tripwire FIFO read, the device driver is presented with a list of endpoints which require attention. This list improves system performance as the device driver is not required to scan a large number of memory locations looking for such endpoints.

Since the device driver is not required to know where the memory locations which have been used for synchronisation are. It is also not required to have any knowledge or take part in the application level communication protocol. All communication protocol processing can be performed by the application and different applications are free to use differing protocols for their own purposes, and one device driver instance may support a number of such applications.

There is also a problem connected with programming a DMA engine that is addressed by an aspect of the invention. Conventional access to DMA engines is moderated either by a single system device driver, which requires (slow) context switches to access, or by virtualisation of the registers by system page fault, also requiring (multiple) context switches. The problem is that it is not safe for a user level application to directly modify the DMA engine registers or a linked list DMA queue, because this must be

done atomically. In most systems, user applications cannot atomically update the DMA queue as they can be descheduled at any moment.

The invention addresses this problem by using hardware FIFO 50 to queue DMA requests from applications. Each application wanting to request DMA transfers sets up a descriptor, containing the start address and the length of the data to be transferred, in its local memory and posts the address of the descriptor to the DMA queue, whose address is common to all applications. This can be arranged by mapping a single page containing the physical address of the DMA queue as a write-only page into the address space of all user applications as they are initialised.

As soon as DMA work queue FIFO 50 is not empty, local bus 55 is not busy and the DMA engine in bridge 12 is also not busy, Master/Target/DMA arbiter 40 grants DMA state machine 51 access to local bus 55. Using the address posted by the application in FIFO 50, state machine 51 then uses bridge 12 to read the descriptor in memory 6 into the descriptor block 52. State machine 51 then posts the start address and length information held in block 52 into the DMA engine in bridge 12.

When the DMA process is complete, bridge 12 notifies state machine 51 of the completion. The state machine then uses data from descriptor block 52 to write back a completion descriptor in memory 6. Optionally, an interrupt can also be raised on microprocessor 5, although a Tripwire may already have been crossed to provide this notification early in order to minimise the delay bringing the relevant application back onto microprocessor 5's run queue. This is shown later in this document.

Should queue 50 be full, then state machine 51 writes a failure code back into the completion field of the descriptor that the application has just attempted to place on the queue. Thus the application does not need to read the status of the NIC in order to safely post a DMA request. All applications can safely share the same hardware posting address, and no time-consuming virtualisation or system device driver process is necessary.

Should any operation take longer than a preset number of PCI cycles, timeout logic 61 is activated to terminate the current cycle and return an interrupt through block 53.

Another aspect of the invention relates to the protocol which is preferably used by the NIC. This protocol uses an address and some additional bits in its header. This allows the transfer of variable length packets with simple routines for Segmentation and Reassembly (SAR) that are transparent to the sending or receiving codes. This is also done without the need to have an entire packet arrive before segmentation, reassembly or forwarding can occur, allowing the data to be put out on the ongoing link immediately. This enables data to traverse many links without significantly adding to the overall latency. The packets may be fragmented and coalesced on each link, for example between the NIC and a host I/O bus bridge, or between the NIC and another NIC. We term this cut-through routing and forwarding. In a network carrying a large number of streams, cut-through forwarding and routing enables small packets to pass through the network without any delays caused by large packets of other streams. While other network physical layers such as ATM also provide the ability to perform cut-through forwarding and routing, they do so at the cost of requiring all packets to be of a fixed small size.

Figure 8 shows an example of how this protocol has been implemented using the 23-bit data transfer capability of HP's GLINK chipset (serialiser 14 and de-serialiser 16). PCI to local bus bridge 12 provides a bus of 32 address/data bits, 4 parity bits and 4 byte-enable bits. It also provides an address valid signal (ADS) which signifies that a burst is beginning, and that the address is present on the address/data bus. The burst continues until a burst last signal (BLAST) is set active, signifying the end of a burst. It provides a read/write signal, and some other control signals that need not be transferred to a remote computer. Figure 8A shows how this protocol is used to transfer an n data word burst 63. The data traffic closely mirrors that used on the PCI bus, but uses fewer signals.

The destination address always precedes each data burst. Therefore, the bursts can be of variable size, can be split or coalesced, by generating fresh address words, or by removing address words where applicable. In the preferred embodiment, sequential data words are destined for sequentially incrementing addresses. However, data words having sequentially decrementing addresses might also be used, or any other pattern of addresses may be used so long as it remains easy to calculate. So far as the endpoints are concerned, exactly the same data is transferred to exactly the same locations. The benefits are that packets can be of any size at all, reducing the overhead of sending an address; packets can be split (and addresses regenerated to continue) by network switches to provide quality of service, and receivers need not wait for a complete packet to arrive to begin decoding work.

Also, the destination address given in the header may be for the 'nth' data word in the burst, rather than for the first, although using the first data word address is preferred.

Figure 8b shows how the protocol of Figure 8a is transcribed onto the G-LINK physical layer. The first word in any packet contains an 18-bit network address. Each word of 63 is split into two words in 64; the lower 16 bits carry high and low addresses or data, corresponding to the address/data bus; the next 4 bits carry either byte enables or parity data. During the address phase, the byte enable field (only 2 bits of which are available, owing to the limitations of G-LINK) is used to carry a 2-bit code indicating read, write or escape packet use. Escape packets are normally used to carry diagnostic or error information between nodes, or as a means of carrying the Xon/Xoff-style protocol when no other data is in transit. The G-LINK nCAV signal corresponds to the ADS signal of 63; nDAV is active throughout the rest of the burst and the combination of nDAV inactive and nCAV inactive signals the end of a burst, or nCAV active indicates the immediate beginning of another burst.

Figure 8c, shows a read data burst 65; this is the same as a write burst 64, except data bit 16 is set to 0. On the outbound request, the data field contains the network address for the read data to be returned to. When the data for a read returns 66, it travels like a

write burst, but is signified by there only being one nCAV active (signifying the network address) along with the first word. An additional bit, denoted FLAG in Figure 8, is used to carry Xon/Xoff style information when a burst is in progress. It is not necessary therefore to break up a burst in order to send an Escape packet containing the Xon/Xoff information. The FLAG bit also serves as an additional end of packet indicator.

In Figure 8c, 67,68 shows an escape packet; after the network address, this travels with 68 or without 67 a payload as defined by data bit 16 in the first word of the burst.

In a full networked implementation, an extra network address word may precede each of these packets. Other physical layer or network layer solutions are possible, without compromise to this patent application, including fibre channel parts (using 8B/10B encoding) and conventional networks such as ATM or even Ethernet. The physical layer only needs to provide some means of identifying data from non-data and the start of one burst from the end of a previous one.

A further aspect of the invention relates to the distribution of hardware around a network. One use of a network is to enable one computer to access a hardware device whose location is physically distant. As an example, consider the situation shown in Figure 9, where it is required to display the images viewed by the camera 70, (connected a frame-grabber card 69) on the monitor which is, in turn, connected to computer 72. The NIC 73 is programmed from Boot ROM 22 to present the same hardware interface as that of the frame-grabber card 69. Computer 72 can be running the standard application program as provided by a third party vendor which is unaware that system has been distributed over a network. All control reads and writes to the frame-grabber 69, are transparently forwarded by the NIC 73, and there is no requirement for an extra process to be placed in the data path to interface between the application running on CPU 74 and the NIC 73. Passive PCI I/O back-plane 71, requires simply a PCI bus clock and arbiter i.e., no processor, memory or cache. These functions can be implemented at very low cost.

The I/O buses are conformant to PCI Local Bus Specification 2.1. This PCI standard supports the concept of a bridge between two PCI buses. It is possible to program the NIC 73 to present the same hardware interface as a PCI bridge between Computer 72 and passive back-plane 71. Such programming would enable a plurality of hardware devices to be connected to back-plane 71 and controlled by computer 72 without the requirement for additional interfacing software. Again, it should be clear that the invention will support both general networking activity and this remote hardware communication, simultaneously using a single network card.

A circular buffer abstraction will now be discussed as an example of the use of the NIC by an application. The circular buffer abstraction is designed for applications which require a producer/consumer software stream abstraction, with the properties of low latency and high bandwidth data transmission. It also has the properties of responsive flow control and low buffer space requirements. Fig. 10 shows a system comprising two software processes, applications 102 and 103, on different computers 100, 101. Application 102 is producing some data. Application 103 is awaiting the production of data and then consuming it. The circular buffer 107, is composed of a region of memory on Computer 101 which holds the data and two memory locations - RDP 106 and WRP 109. WRP 109 contains the pointer to the next byte of data to be written into the buffer, while RDP 106 contains the pointer to the last byte of data to be read from the buffer. When the circular buffer is empty, then WRP is equal to $RDP + 1$ modulo wrap-around of the buffer. Similarly, the buffer is full when WRP is equal to $RDP - 1$. There are also private values of WRP 108 and RDP 111 in the caches of computer 100 and computer 101 respectively. Each computer 100, 101 may use the value of WRP and RDP held in its own local cache memory to compute how much data can be written to or read from the buffer at any point in time, without the requirement for communication over the network.

When the circular buffer 107 is created, the producer sets up a Tripwire 110, which will match on a write to the RDP pointer 106, and the consumer sets up a Tripwire 113, which will match on a write to the WRP pointer 109.

If consumer application 103 attempts to read data from the circular buffer 107, it first checks to see if the circular buffer is empty. If so, application 103 must wait until the buffer is not empty, determined when WRP 109 has been seen to be incremented. During this waiting period, application 103 may either block, requesting an operating system reschedule, or poll the WRP 109 pointer.

If producer application 102 decides to write to the circular buffer 107, it may do so while the buffer is not full. After writing some data, application 102 updates its local cached value of WRP 108, and writes the updated value to the memory location 109, in computer 101. When the value of WRP 109, is updated, the Tripwire 113, will match as has been previously described.

If consumer application 103 is not running on CPU 118 when some data is written into the buffer and Tripwire 113 matches, NIC 115 will raise a hardware interrupt 114. This interrupt causes CPU 118 to run device driver software contained within operating system 118. The device driver will service the interrupt by reading the tripwire FIFO 42 on NIC 115 and determine from the value read, the system identifier for application 103. The device driver can then request that operating system 118, reschedule application 103. The device driver would then indicate that the tripwire 113 should not generate a hardware interrupt until application 103 has been next descheduled and subsequently another Tripwire match has occurred.

Note that the system identifier for each running application is loaded into internal registers 49, each time the operating system reschedules. This enables the NIC to determine the currently running application, and so make the decision whether or not to raise a hardware interrupt for a particular application given a Tripwire match.

Hence, once consumer application 103 is again running on the processor further writes to the circular buffer 107, by application 102, may occur without triggering further hardware interrupts. Application 103 now reads data from the circular buffer 107. It can read data until the buffer becomes empty (detected by comparing the values of RDP and

WRP 111,109). After reading, application 102 will update its local value of RDP 111 and finally writes the updated value of RDP to memory location 106 over the network.

If producer application 102 had been blocked on a full buffer, this update of RDP 106 would generate a Tripwire match 110, resulting in application 102, being unblocked and able to write more data into the buffer 107.

In normal operation, application 102 and application 103 could be operating on different parts of the circular buffer simultaneously without the need for mutual exclusion mechanisms or Tripwire.

The most important properties of the data structure are that the producer and the consumer are able to process data without hindrance from each other and that flow control is explicit within the software abstraction. Data is streamed through the system. The consumer can remove data from the buffer at the same time as the producer is adding more data. There is no danger of buffer over-run, since a producer will never transmit more data than can fit in the buffer.

The producer only ever increments WRP 108, 109 and reads RDP 106, and the consumer only ever increments RDP 106, 111, and reads WRP 109. Inconsistencies in the values of WRP and RDP seen by either the producer or consumer either cause the consumer to not process some valid data (when RDP 106 is inconsistent with 111), or the producer to not write some more data (when WRP 109 is inconsistent with 108), until the inconsistency has been resolved. Neither of these occurrences cause incorrect operation or performance degradation so long as they are transient.

It should also be noted that on most computer architectures, including the Alpha AXP and Intel Pentium ranges, computer 100 can store the value of the RDP 106 pointer in its processor cache, since the producer application 102 only reads the pointer 106. Any remote writes to the memory location of the RDP pointer 106 will automatically invalidate the copy in the cache causing the new value to be fetched from memory. This

process is automatically carried out and managed by the system controller 8. In addition, since computer 101 keeps a private copy of the RDP pointer 111 in its own cache, there is no need for any remote reads of RDP pointer values during operation of the circular buffer. Similar observations can also be made for the WRP pointer 109 in the memory of computer 101 and the WRP pointer 108 in the cache of computer 100. This feature of the buffer abstraction ensures that high performance and low latency are maintained. Responsive application level flow-control is possible because the cached pointer values can be exposed to the user-level applications 102, 103.

A further enhancement to the above arrangement can be used to provide support for applications which would like to exchange data in discrete units. As shown in Fig. 11, and in addition to the system described in Fig. 10. The system maintains a second circular buffer 127, of updated WRP 129 values corresponding to buffer 125. This second buffer 127 is used to indicate to a consumer how much data to consume in order that data be consumed in the same discrete units as it were produced. Note that circular buffer 125 contains the data to be exchanged between the applications 122 and 123.

The producer, application 122 writes data into buffer 125, updating the pointer WRP 129, as previously described. Once data has been placed in buffer 125, application 122 then writes the new value of the WRP 129 pointer into buffer 127. At the same time it also manipulates the pointer WRP 131. If either of these write operations does not complete then the application level write operation is blocked until some data is read by the consumer application 123. The Tripwire mechanism can be used as previously described, for either application to block on either a full or empty buffer pair.

The consumer application 123 is able to read from both buffers 125 and 127, in the process updating the RDP pointers 133, 135 in its local cache and RDP pointers 124, 126 over the network in the manner previously described. A data value read from buffer 127 indicates an amount of data, which had been written into buffer 125. This value may be used by application level or library software 123, to consume data from buffer

125 in the same order and by the same discrete amounts as it were produced by application 122.

The NIC can also be used to directly support a low latency Request/Response style of communication, as seen in client/server environments such as Common Object Request Broker Architecture (CORBA) and Network File System (NFS) as well as transactional systems such as databases. Such an arrangement is shown in Fig.12, where application 142 on computer 140 acts as a client requesting service from application 143 on computer 141, which acts as a server. The applications interact via memory mappings using two circular buffers 144 and 145, one contained in the main memory of each computer. The circular buffers operate as previously described, and also can be configured to transfer data in discrete units as previously described.

Application 142, the client, writes a request 147 directly into the circular buffer 145, via the memory mapped connection(s), and waits for a reply by waiting on data to arrive in circular buffer 144. Most Request/Response systems use a process known as marshalling to construct the request and use an intermediate buffer in memory of the client application to do the marshalling. Likewise marshalling is used to construct a response, with an intermediate buffer being required in the memory of the server application. Using the present invention, marshalling can take place directly into the circular buffer 145 of the server as shown. No intermediate storage of the request is necessary at either the client or server computers 140, 141.

The server application 143 notices the request (possibly using the Tripwire mechanism) and is able to begin unmarshalling the request as soon as it starts to arrive in the buffer 145. It is possible that the server may have started to process the request 149 while the client is still marshalling and transmitting, thus reducing latency in the communication.

After processing the request, the server writes the reply 146 directly into buffer 144, unblocking application 142 (using the Tripwire mechanism), which then unmarshalls and processes the reply 148. Again, there is no need for intermediate storage, and

unmarshalling by the client may be overlapped with marshalling and transmission by the server.

A further useful and novel property of a Request/Response system built using the present invention, is that data may be written into the buffer both from software running on a CPU, or any hardware device contained in the computer system. Fig. 15 shows a Request/Response system which is a file serving application. The client application 262 writes a request 267 for some data held on disks controlled by 271. The server application 263 reads 269 and decodes the request from its circular buffer 265 in the manner previously described. It then performs authentication and authorisation on the request according to the particular application.

If the request for data is accepted, the server application 263 uses a two-part approach to send its reply. Firstly, it writes, into the circular buffer 264, the software generated header part of the reply 266. The server application 263 then requests 273 that the disk controller 271 send the required data part of the reply 272 over the network to circular buffer 264. This request to the disk controller takes the form of a DMA request, with the target address being an address on I/O bus 270 which has been mapped onto circular buffer 264. Note that the correct offset is applied to the address such that reply data 272 from the disk is placed immediately following the header data 266.

Before initiating the request 273, the server application 263 can ensure that sufficient space is available in the buffer 264 to accept the reply data. Further, it is not necessary for the server application 263 to await the completion request 273. It is possible for the client application 262 to have set a Tripwire 274 to match once the reply data 272 has been received into buffer 264. This match can be programmed to increment the WRP pointer associated with buffer 264, rather than requiring application 263 to increment the pointer as previously described. If a request fails, then the client application 262 level timeout mechanism would detect and retry the operation.

It is also possible for the client application 262 to arrange that reply data 272 be placed in some other data structure, (such as a kernel buffer-cache page), through manipulation of 169 and 167 as described later. This is useful when 264 is not the final destination of the reply data, so preventing a final memory copy operation by the client. Server application 263 would be unaware of this client side optimisation.

By use of this mechanism, the processing load on the server is reduced. The requirement for the server application to wait for completion of its disk requests is removed. The requirement for high bandwidth streams of reply data to pass through the server's system controller, memory, cache or CPU is also removed.

As previously stated, the NIC of the present invention could be used to support the Virtual Interface Architecture (VIA) Standard. Fig. 13 shows two applications communicating using VIA. Application 152 sends data to application 153, by first writing the data to be sent into a region of its memory, shown as block 154. Application 152 then builds a transmit descriptor 156, which describes the location of block 154 and the action required by the NIC (in this case data transmission). This descriptor is then placed onto the TxQueue 158, which has been mapped into the user-level address-space of application 152. Application 152 then finally writes to the doorbell register 160 in the NIC 162 to notify the NIC that work has been placed on the TxQueue 158.

Once the doorbell register 160 has been written, the NIC 162 can determine, from the value written, the address in physical memory of the activated TxQueue 158. The NIC 162 reads and removes the descriptor 156 from the TxQueue 158, determines from the descriptor 156, the address of data block 154 and invokes a DMA 164 engine to transmit the data contained in block 154. When the data is transmitted 168, the NIC 162 places the descriptor 156 on a completion queue 166, which is also mapped into the address space of application 152, and optionally generates a hardware interrupt. The application 152 can determine when data has been successfully sent by examining queue 166.

When application 153 is to receive data, it builds a receive descriptor 157 describing where the incoming data should be placed, in this case block 155. Application 153 then places descriptor 157 onto RxQueue 159, which is mapped into its user-level address-space. Application 153 then writes to the doorbell register 161 to indicate that its RXQueue 159 has been activated. It may choose to either poll its completion queue 163, waiting for data to arrive, or block until data has arrived and a hardware interrupt generated.

The NIC 165 in computer 151 services the doorbell register 161 write by first removing the descriptor 157 from the RxQueue 159. The NIC 165 then locates the physical pages of memory corresponding to block 155 and described by the receive descriptor 157. The VIA standard allows these physical pages to have been previously locked by application 153 (preventing the virtual memory system moving or removing the pages from physical memory). However, the NIC is also capable of traversing the page-table structures held in physical memory and itself locking the pages.

The NIC 165 continues to service the doorbell register write and constructs a Translation Look-aside (TLB) entry 167 located in SRAM 23. When data arrives corresponding to a particular VIA endpoint, the incoming address matches an aperture 169 in the NIC, which has been marked as requiring a TLB translation. This translation is carried out by state machine 46 and determines the physical memory address of block 155.

The TLB translation, having been previously set up, occurs with little overhead and the data is written 175 to appropriate memory block 155. A Tripwire 171 will have been arranged (when the TLB 167 entry was constructed) to match when the address range corresponding to block 155 is written to. This Tripwire match causes the firmware 173 (implemented in state machine 51) to place the receive descriptor 157 onto completion queue 163 to invalidate the TLB mapping 167 and optionally generate an interrupt. If the RxQueue 159 has been loaded with other receive descriptors, then the next descriptor is taken and loaded into the TLB as previously described. If application 153

is blocked waiting for data to arrive, the interrupt generated will result, (after a device driver has performed a search of all the completion queues in the system), in application 153 being re-scheduled. If there is no TLB mapping for the VIA Aperture addresses, or the mapping is invalid, an error is raised using an interrupt. If the NIC 165 is in the process of reloading the TLB 167 when new data arrives, then hardware flow control mechanism 31 is used to control the data until a path to the memory block in computer 151 has been completed.

As an optional extension to the VIA standard, the NIC could also respond to Tripwire match 171 by placing an index on Tripwire FIFO 42, which could enable the device driver to identify the active VIA endpoint without searching all completion queues in the system.

This method can be extended to provide support for I20 and the forthcoming Next Generation I/O (NGIO) standard. Here, the transmit, receive and completion queues are located on the NIC rather than in the physical memory of the computer, as is currently the case for the VIA standard.

As mentioned previously, another aspect of this invention is its use in providing support for the outbound streaming of data through the NIC. This setup is described in Fig. 14. It shows a Direct Memory Access (DMA) engine 182 on the NIC 183, which has been programmed in the manner previously described by a number of user-level applications 184. These applications have requested that the NIC 183 transfer their respective data blocks 181 through the NIC 183, local bus 189, fibre-optic transceiver 190 and onto network 200. After each application has placed its data transfer request onto the DMA request queue 185, it blocks, awaiting a re-schedule, initiated by device driver 187. It can be important that the system maintains fair access between a large number of such applications, especially under circumstances where an application requires a strict periodic access to the queue, such as an application generating a video stream.

Data transferred over the network by the DMA engine 182, traverses local bus 189, and is monitored by the Tripwire unit 186. This takes place in the same manner as for received data, (both transmitted and received data pass through the NIC using the same local bus 55).

Each application, when programming the DMA engine 182 to transmit a data block, also constructs a Tripwire which is set to match on an address in the data block. The address to match could indicate that all or a certain portion of the data has been transmitted. When this Tripwire fires and causes a hardware interrupt 188, the device driver 187 can quickly determine which application should be made runnable. By causing a system reschedule, the application can be run on the CPU at the appropriate moment to generate more DMA requests. Because the device driver can execute at the same time that the DMA engine is transferring data, this decision can be made in parallel to data transfer operations. Hence, by the time that a particular application's data transfer requests have been satisfied, the system can ensure that the application be running on the CPU and able to generate more requests.

Figure 16 illustrates a generalised apparatus or arrangement for synchronising an end-point application using a tripwire. An end-point is a final destination for an information stream and is the point at which processing of the information takes place. Examples of end-points include a web, a file, a database server and hardware devices such as a disk or graphics controller. An end-point may be running an operating system and a number of data processing applications and these are referred to as end-point applications. Thus, examples of end-point applications include an operating system or a component thereof, a network protocol stack, and any application-level processing. Arrangements such as network switches and routers do not constitute end-points or end-point applications because their purpose is to ensure that the information is delivered elsewhere.

The arrangement comprises a computer 300 which is optionally connected to other computers 301 and 302 via a network 303. The computer 300 comprises a program

memory (illustrated by way of example only as a read only memory (ROM) 305) which contains a program for controlling the computer to synchronise the end-point application in accordance with an address-based event in an information stream on an information pathway 307, such as a bus, within the computer. The information stream may be wholly within the computer, for example from another application performed by the computer 300, or may be from a remote source, such as from the network 303.

The bus 307 is connected to a memory 308 in the end-point application 306, which also comprises a code generator 309 and an action generator 310. The code generator 309 supplies codes to a comparator which is illustrated as a content addressable memory (CAM) 311. The CAM 311 has another input connected to the bus 307 and is arranged to perform a comparison between each entry in the CAM and the information stream on the bus 307. When a match is found, the CAM sends a signal to the action generator 310 which performs an action which is associated with an address-based event in the information stream.

In a typical example of use of the synchronising arrangement, the end-point application 306 sets a tripwire, for example to be triggered when data relating to an end-point address or range of end-point addresses in the memory 308 are present on the bus 307. The code generator 309 supplies a code which is written into the CAM 311 and which comprises the destination memory address of the data or possibly part of this address, such as the most significant bits when a range of addresses is to be monitored. It is also possible to enter a code which represents not only the address or range of addresses but also part or all of one or more items of data which are expected in the information stream. The CAM 311 compares the address of each data burst on the bus 307, and possibly also at least some of the data of each burst, with each code stored in the CAM 311 and supplies a signal to the action generator 310 when a match is found. The action generator 310 then causes the appropriate action to be taken within the end-point application 306. This may be a single action, several actions, or one or more specific actions which are determined not only by the triggering of the tripwire but also by the

data within the information stream, for example arriving at the appropriate location or locations in the memory 308.

As mentioned hereinbefore, the information stream 307 may be wholly internal to the computer 300 and an example of this is an application-to-application stream of information where both applications are running, for example alternately, on the computer 300. However, the information stream may be partly or wholly from outside the computer 300, as illustrated by the broken line connection from the bus 307 to the network 303. Thus, the information stream may be from a switch fabric, a network, or a plurality of sources. A switch fabric is a device which has a plurality of inputs and outputs and which is capable of forwarding data from each input to the appropriate output according to routing information contained within the data. A switch fabric may alternatively be wholly contained within the computer. The information stream preferably has a data burst arrangement as described hereinafter and, in the case of a plurality of sources, the data bursts may arrive from any of the sources at any time, which amounts to multiplexing.

Figure 17 shows an arrangement which illustrates two possible modifications to the arrangement shown in Figure 16. In this case, the bus 307 is connected to an input/output bus 312 of the end-point application 306 within the computer 300. This represents an example of a hardware end-point for the information stream but other types of hardware end-points are possible, such as active controllers, and may be located "outside" the application 306. An example of an active controller is a disk controller.

The arrangement shown in Figure 17 also differs from that shown in Figure 16 in that the tripwire may be triggered by an address-based event in the information stream on the bus 307 which does not exactly match any of the codes stored in the CAM 311. Instead, the information from the information stream on the bus 307 first passes through a process 313 before being supplied to the CAM for comparison with each of the stored codes.

One application of this is for the case where the information stream comprises packets or bursts of data starting with an address, for example corresponding to an address in the memory 308 to which the first item of data after the address in the packet or burst is allocated. Subsequent items of data are to be allocated to consecutive addresses, for example such that each item of data in the burst is to be allocated to the next highest address location after the preceding data item. Thus, the address at the start of each burst relates to the first data item and the following data item addresses can be inferred by incrementing the address upon the arrival of the second and each subsequent item of data.

The application 306 can cause the code generator 309 to store in the CAM 311 a code which corresponds to an implied address in the actual information stream appearing on the bus 307. The process 313 detects the address at the start of each data burst and supplies this to the CAM 311 with the arrival of the first data item. As each subsequent data item of the same burst arrives, the process 313 increments the address and supplies this to the CAM 311. This allows a tripwire to be triggered when, for example a data item having an implied address is present on the bus 307 because the CAM can match the corresponding stored code with the address supplied by the process 313.

As mentioned hereinbefore, the action generator 310 can cause any one or more of various different actions to be triggered by the tripwire. The resulting action may be determined by which tripwire has been triggered i.e. which code stored in the CAM 311 has been matched. It is also possible for the action to be at least partly determined by the data item which effectively triggered the tripwire. Any action may be targetted at the computer containing the tripwire or at a different computer. Various possible actions are described hereinafter as typical examples and may be performed singly or in any appropriate combination for the specific application and may be targetted at the computer containing the tripwire or at a different computer.

Figure 18 illustrates the action generator 310 raising an interrupt request IRQ and supplying this to the interrupt line of a central processing unit (CPU) 320 of the

computer 300. Figure 19 illustrates the action generator 310 setting a bit in a bitmap 321, for example in the memory 308. These two actions may be used independently of each other or together. For example, the action generator may raise an interrupt request if an application which requires data corresponding to the tripwire is not currently running but is runnable; for example it has not exhausted its time-slice. Otherwise, for example if the application is awaiting rescheduling, the relevant bit in the bitmap 321 may be set. The operating system may periodically check the bitmap 321 for changes and, as a result of the arrival of the relevant data for an application which is presently not running, may decide to reschedule or wakeup the application.

Figure 20 illustrates another type of action which may be performed as a result of detection of the address-based event. In this example, a counter 322, for example whose count is stored within the memory 308, is incremented in response to triggering of the tripwire. Incrementing may take place as a result of any tripwire being triggered or only by one or more specific tripwires depending on the specific application.

Figure 21 illustrates another action which is such that, when the or the appropriate tripwire is triggered, a predetermined value "N" is written to a location "X" shown at 323 as being in the memory 308 (or being mapped thereto).

Figure 22 illustrates another combination of actions which may be used to indicate that an application should be awakened or rescheduled. When a tripwire is triggered, an interrupt request is supplied to the CPU 320 and a "runnable bit" for a specific application is set at location 324 in the memory 308. The operating system of the computer 300 responds to the interrupt request by waking up or rescheduling the application whose runnable bit has been set.

Figure 23 illustrates an action which modifies entries in the CAM 311 in response to triggering of a tripwire. Any form of modification is possible. For example, the code which triggers the tripwire may be deleted if no further tripwires are required for the same address-based event. As an alternative, the code may be modified so as effectively to set a different but related tripwire. A further possibility is to generate a

completely new code and supply this to the CAM 311 in order to set a new unrelated tripwire.

Figure 24 illustrates the format of a data burst, a sequence of which forms the information stream on the bus 307. The data burst comprises a plurality of items which arrive one after the other in sequence on the bus. The first item is an address $A(n)$ which is or corresponds to the end-point address, for example in the memory 308, for receiving the subsequent data items. This address is the actual address n of the first data item D_1 of the burst, which immediately follows the address $A(n)$. The subsequent data items D_2, D_3, \dots, D_p arrive in sequence and their destination addresses are implied by their position within the burst relative to the first data item D_1 and its address n . Thus, the second data item D_2 has an implied address $n + 1$, the third data item D_3 has an implied address $n + 2$ and so on. Each data item is written or supplied to the implied address as its destination address.

This data burst format may be used to fragment and coalesce bursts as the data stream passes through a forwarding unit 330, such as a network interface card or a switch, of an information pathway. For example, the forwarding unit can start to transmit a burst as soon as the first data item has arrived and does not have to wait until the whole data burst has arrived.

Figure 25 illustrates an example of this in which an interruption in the data burst occurs. The forwarding unit 330 has already started transmission of the burst and the first r data items 331 together with the burst address have already been forwarded. The remainder 332 of the burst has not yet arrived and the forwarding unit 330 terminates forwarding or transmission of that burst.

When the remainder 332 of the burst starts to arrive, the forwarding unit 330 recalculates the destination address $A(r+1)$ for the remainder of the burst and inserts this in front of the data item D_{r+1} . This is transmitted as a further burst 333 as illustrated in Figure 26.

This technique may be used even when the whole burst is available for forwarding by the forwarding unit 330. For example, the forwarding unit 330 may terminate transmission of a particular burst before completion of transmission for reasons of arbitration between a number of competing bursts or for flow control reasons. Thus, individual data bursts can be forwarded in tact or can be sent in two or more fragments as necessary or convenient and all such bursts are treated as valid bursts by any subsequent forwarding units.

Figure 27 illustrates an alternative situation in which the forwarding unit has an internal buffer 335 which contains first and second bursts 336 and 337. In this case, the implied address of the first data item $D_n + 1$ of the second burst 337 immediately follows the implied address of the last data item D_n of the first burst 336. The forwarding unit checks for such situations and, when they are found, coalesces the first and second bursts into a coalesced burst 338 as shown in the lower part of Figure 27. The forwarding unit then transmits a single contiguous burst, which saves the overhead of the excess address information (which is deleted from the second burst). Any subsequent forwarding units then treat the coalesced burst 338 as a single burst. The format of the data burst allows such fragmentation or merging of bursts to take place. This in turn allows forwarding units to transmit data as soon as it arrives so as to reduce or minimise latency. Also, bursts of any length or number of data items can be handled which improves the flexibility of transmission of data.

Figure 28 illustrates an example of communication between an application, whose address space is shown at 340, and remote hardware 341 via a network 303 such that the network 303 is "transparent" or "invisible" to each of the application and the remote hardware 341. The address space 340 contains mapped configuration data and registers of the remote hardware as indicated at 342. This is mapped onto the system input/output bus 343 to which a network interface card 344 is connected. The network interface card 344 is loaded with configuration and register data corresponding to the remote hardware 341. All application requests are forwarded over the network 303

transparently to the remote hardware 341 so that the remote hardware appears as though it is local to the application and the network 303 is invisible.

The remote hardware 341 is connected to a passive input/output bus 345 which is provided with a network interface card 346 for interfacing to the network 303. The configuration and registers of the remote hardware are illustrated at 347 and are mapped ultimately to the region 342 of the address space 340 of the application. Again, the network is invisible to the remote hardware 341 and the remote application appears to be local to it.

When the application sends a request to the remote hardware 341, for example requesting that the remote hardware supply data to be used in or processed by the application, this is written in the space 342 which is mapped to the system input/output bus 343. The network interface card 344 sends read/write requests over the network 303 to the card 346, which supplies these via the passive input/output bus 345 to the remote hardware 341. Viewed from the remote hardware 341, the bus 345 appears equivalent to the bus 343.

The remote hardware 341 may supply an interrupt and/or data for the application to the bus 345. Again, the network interface card 346 sends this via the network 303 to the card 344. The network interface card 344 supplies an interrupt request to the computer running the application and writes the data on behalf of the remote hardware to the space 342 in the address space 340 of the application. Thus, to the application, the remote hardware 341 appears to be connected directly to the bus 343.

Although implementations of tripwires have been described in detail hereinbefore with reference to the tripwire unit 1 shown in Figure 29 associated with the network interface card 350, tripwires may be implemented at other points in a system as illustrated by tripwire units 2 to 5 in Figure 29. The system comprises a disk controller 351 connected to an input/output bus 307b and the tripwire unit 2 is implemented as part of the disk controller 351. Such an arrangement allows tripwire operations to inform

applications of any characteristic data transfer to or from the disk controller 351. Such an arrangement is particularly useful where the controller 351 is able to transfer data to and from a non-contiguous memory region corresponding to user-level buffers of an application. This allows data transfer and application level notification to be achieved without requiring hardware interrupts or kernel intervention.

The tripwire unit 3 is associated with a system controller 352 connected to a host bus 307a and the input/output bus 307b. Such an arrangement allows tripwire operations to inform applications of any characteristic data transfer to or from any device in the computer system. This includes hardware devices, such as the disk controller 351 and the network interface card 350, and, in the case of a system employing several CPUs, enables an application running on one of the CPUs to synchronise on a data transfer to or from an application running on another of the CPUs. Similarly, a tripwire may be used for synchronisation between applications running on the same CPU. This reduces the need for other mechanisms such as spin locks where both applications are required to operate in lock-step with the data transfer.

Tripwire units 4 and 5 are implemented in the CPU 320 or the memory 308. This is generally equivalent to the tripwire unit 3, where all data transfers in the system can be monitored. However, the tripwire unit 4 may monitor data written by an application to cache, which may not appear on the host bus 307a.

CLAIMS

1. A method of synchronising an end-point application in a computer, comprising the steps of:
 - (a) generating and storing at least one code whose purpose is to associate an action with an address-based event in an information stream, which comprises data and associated memory addresses, on an information pathway within the computer;
 - (b) comparing the generated code with each of at least the associated addresses to detect the address-based event; and
 - (c) performing the associated action in response to detection of the address-based event.
2. A method as claimed in claim 1, in which the step (b) comprises comparing the at least one generated code only with each of the associated addresses.
3. A method as claimed in claim 1, in which the step (b) comprises comparing the at least one generated code with each of the associated addresses and at least part of the data.
4. A method as claimed in any one of claims 1 to 3, in which the associated addresses are processed before being compared with the at least one generated code in the step (b).
5. A method as claimed in any one of claims 1 to 4, in which the data have inferred addresses.
6. A method as claimed in claim 5 when dependent on claim 4, in which: the information stream comprises a series of data bursts, each of which comprises an associated address and consecutive items of data; the processing step comprises reading the associated address and incrementing the read address upon the arrival of each item after the first item of a burst; and the step (b) comprises comparing the at least one generated code with the associated address processed by the processing step.

7. A method as claimed in any one of claims 1 to 6, in which the information pathway is a computer bus.
8. A method as claimed in any one of claims 1 to 7, in which the information pathway is a switch fabric.
9. A method as claimed in any one of claims 1 to 8, in which the information stream is from a network of computers.
10. A method as claimed in any one of claims 1 to 9, in which the information stream is from a plurality of sources and is multiplexed.
11. A method as claimed in any one of claims 1 to 8, in which the information stream is wholly within the computer.
12. A method as claimed in any one of claims 1 to 11, in which each associated address represents a memory location or range of locations at the end-point application.
13. A method as claimed in any one of claims 1 to 12, in which the step (a) is performed by at least one application of the computer.
14. A method as claimed in claim 13, in which the at least one application includes the end-point application.
15. A method as claimed in any one of claims 1 to 14, in which the step (b) is performed by a content-addressable memory.
16. A method as claimed in any one of claims 1 to 15, in which the associated action comprises a plurality of associated actions.
17. A method as claimed in any one of claims 1 to 16, in which the associated action comprises raising an interrupt for the end-point application.

18. A method as claimed in claim 17, in which the interrupt is raised only if the end-point application is not running.
19. A method as claimed in any one of claims 1 to 18, in which the associated action comprises setting a bit in a bitmap which is readable by the end-point application.
20. A method as claimed in any one of claims 1 to 19, in which the associated action comprises incrementing an event counter.
21. A method as claimed in any one of claims 1 to 20, in which the associated action comprises writing a predetermined value to a predetermined memory location.
22. A method as claimed in any one of claims 1 to 21, in which the associated action comprises deleting the at least one generated code.
23. A method as claimed in any one of claims 1 to 21, in which the associated action comprises modifying the at least one generated code.
24. A method as claimed in any one of claims 1 to 23, in which the associated action comprises generating and storing at least one further code.
25. A method as claimed in any one of claims 1 to 24, in which the associated action comprises rescheduling the end-point application.
26. A method as claimed in any one of claims 1 to 24, in which the end-point application is suspended after the at least one code is generated and the associated action comprises waking up the end-point application.
27. A computer program for controlling a computer to perform a method as claimed in any one of claims 1 to 26.
28. A storage medium containing a program as claimed in claim 27.

29. A computer programmed by a program as claimed in claim 27.

30. A method of synchronising between a sending application on a first computer and a receiving application on a second computer, each computer having a main memory, and at least one of the computers having an asynchronous network interface, comprising the steps of:

providing the asynchronous network interface with a set of rules for directing incoming data to memory locations in the main memory of the second computer;

storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the applications; receiving, at the network interface, a data stream being transferred between the applications;

comparing at least part of the data stream received with the stored triggering values;

if any compared part of the data stream matches any triggering value, indicating that the triggering value has been matched; and

storing the data received in the main memory of the second computer at one or more memory location(s) in accordance with the said rules.

31. A method according to claim 30, in which the step of providing the asynchronous network interface with a set of rules comprises the step of establishing a mapping between information contained within the incoming data stream and one or more memory location(s) of the main memory of the second computer.

32. A method according to claim 31, in which the asynchronous network interface is a memory mapped network interface, and in which the step of providing the memory mapped network interface with a set of rules comprises the step of establishing a mapping between addresses contained within the incoming data stream and one or more memory location(s) of the main memory of the second computer.

33. A method according to any of claims 30 to 32, further comprising storing in the asynchronous network interface an action, corresponding to each triggering value,

which is to be carried out, in the event that the triggering value is matched, to indicate that the triggering value has been matched.

34. A method according to any of claims 30 to 33, comprising the step of sending an interrupt when a triggering value matches.

35. A method according to any of claims 30 to 34, comprising the step of changing the value of a counter when a triggering value is matched.

36. A method according to any one of claims 30 to 35, in which the triggering value(s) comprise(s) address data, and the part of the data stream compared with the stored triggering value(s) comprises address data.

37. A method according to any one of claims 30 to 36, wherein the step of storing a triggering value is initiated by an application on one of the computers writing a triggering value to a memory location in the local control aperture within the address space of the network interface.

38. A method according to any one of claims 30 to 37, comprising the steps of accessing the main memory of the sending application, and outputting data therefrom.

39. A method according to any one of claims 30 to 38, comprising the step of mapping each physical destination address of the data being sent to a virtual memory address on a sending computer.

40. A method according to any one of claims 30 to 39, both computers having an asynchronous network interface, comprising the step of sending the data stream from the sending network interface to the receiving network interface.

41. A method according to claim 40, comprising the step of mapping each virtual address of the received data stream to a physical address memory location of the main memory of the receiving computer.

42. A method according to any one of claims 30 to 41, comprising the step of writing the transferred data to the main memory of the receiving computer.

43. A method according to any one of claims 30 to 42, each computer having a network interface also having an I/O bus, the method comprising the step of providing the network interface with a local bus, and a bridge for interfacing between the local bus and the I/O bus of the computer.

44. A method according to claim 43, comprising the step of loading the bridge with predetermined configuration data.

45. An asynchronous network interface, for use in a host computer having a main memory and being connected to a network, the interface comprising:

means for storing a set of rules for directing incoming data to memory locations in the main memory of the host computer;

a memory for storing one or more triggering value(s), each value representing a state of a data transfer between two or more applications in the computer network;

a receiver for receiving a data stream being transferred between two or more applications in the computer network;

comparison means for comparing at least part of the data stream received by the network interface with the stored triggering values; and

a memory for storing information identifying any matched triggering values.

46. An asynchronous network interface according to claim 45, in which the set of rules comprises a memory mapping.

47. An asynchronous network interface according to claim 45 or 46, further comprising means for performing an action corresponding to a matched triggering value.

48. An asynchronous network interface according to claim 45, 46 or 47, further comprising a local bus.

49. An asynchronous network interface according to claim 48, the host computer having an I/O bus, the interface further comprising a bridge for interfacing between the I/O bus of the computer and the local bus of the network interface.
50. An asynchronous network interface according to any of claims 45 to 49, wherein the comparison means comprises a content-addressable memory.
51. An asynchronous network interface according to claim 50, wherein the comparison means comprises two or more content-addressable memories which are arranged so as to conduct a pipelined comparison of the data stream received by the network interface.
52. An asynchronous network interface according to any of claims 45 to 51, further comprising receive and transmit serialisers.
53. An asynchronous network interface according to any of claims 45 to 52, comprising a memory for storing configuration data for the bridge.
54. An asynchronous network comprising two or more computers each having an asynchronous network interface according to any of claims 45 to 53.
55. A method of passing data between an application on a first computer and remote hardware within a second computer or on a passive backplane, the first computer having a main memory and an asynchronous network interface, the method comprising the steps of:
- providing the asynchronous network interface with a set of rules for directing incoming data to memory or I/O location(s) of the remote hardware;
 - storing in the network interface one or more triggering value(s), each triggering value representing a state of a data transfer between the application and the hardware;
 - receiving, at the network interface, a data stream being transferred between the application and the hardware;

comparing at least part of the data stream received with the stored triggering value(s);

indicating that a triggering value has been matched, if any compared part of the data stream matches a triggering value;

and, when a data stream is being passed from the first computer to the remote hardware, storing data received by the remote hardware in memory or I/O location(s) of the remote hardware in accordance with the said rules; and,

when a data stream is being transferred from the remote hardware to the first computer, storing the data received in the main memory of the first computer at one or more memory location(s) in accordance with the said rules.

56. A method according to claim 55, in which the step of providing the asynchronous network interface with a set of rules comprises the step of establishing a mapping between information contained within the incoming data stream and one or more memory or I/O location(s) of the receiving computer or hardware.

57. A method according to claim 56, in which the asynchronous network interface is a memory mapped network interface, and in which the step of providing the memory mapped network interface with a set of rules comprises the step of the first computer establishing a mapping, either locally or remotely, between addresses contained within the incoming data stream and one or more memory or I/O location(s) of the receiving computer or hardware.

58. A method according to any of claims 55 to 57, further comprising storing in the asynchronous network interface an action, corresponding to each triggering value, which is to be carried out, in the event that the triggering value is matched, to indicate that the triggering value has been matched.

59. A method according to any of claims 55 to 58, comprising the step of sending an interrupt when a triggering value matches.

60. A method according to any of claims 55 to 59, comprising the step of changing the value of a counter when a triggering value matches.
61. A method according to any of claims 55 to 60, in which the triggering value(s) comprise(s) address data, and the part of the data stream compared with the stored triggering value(s) comprises address data.
62. A method according to any of claims 55 to 61, wherein the step of storing a triggering value is initiated by an application on a computer writing a triggering value to a memory location in the local control aperture within the address space of the network interface.
63. A method according to any of claims 55 to 62, comprising the steps of accessing the main memory of the application, and outputting data therefrom.
64. A method according to any of claims 55 to 63, comprising the step of mapping each physical destination address of the data being sent, to a virtual memory address on a computer.
65. A method according to any of claims 55 to 64, both computers having an asynchronous network interface, comprising the step of sending the data stream from the sending network interface to the receiving network interface.
66. A method according to any of claims 55 to 64, comprising the step of mapping each virtual address of the received data stream to a physical memory address or I/O location of the receiving computer or remote hardware.
67. A method according to any of claims 55 to 66, comprising the step of writing the transferred data to the main memory of the receiving computer.
68. A method according to any of claims 55 to 67, each computer or passive backplane having a network interface also having an I/O bus, the method comprising

the step of providing each network interface with a local bus, and a bridge for interfacing between the local bus and the I/O bus of the computer or passive backplane.

69. A method according to claim 68, comprising the step of loading the bridge with predetermined configuration data.

70. A method according to claim 69, in which the configuration data includes configuration data relating to the remote hardware.

71. A method according to any of claims 55 to 70, each computer and/or passive backplane having an I/O bus, the method further comprising the steps of:

loading the network interface of one of the computer(s) and/or of the passive backplane with data for configuring it to capture one or more predefined interrupt signal(s) on the I/O bus of that computer or passive backplane;

transferring a captured interrupt signal over the network to a network interface of another computer or passive backplane; and

loading the network interface of one of the computer(s) or of the passive backplane to assert one or more predefined interrupt signal(s) on the I/O bus of that computer or passive backplane, on receipt of the said transferred captured interrupt signal.

72. A method of arranging data transfers from one or more applications on a computer, the computer having a main memory, an asynchronous network interface, and a Direct Memory Access (DMA) engine having a request queue address common to all the applications, comprising the steps of:

the application requesting the network interface to store one or more triggering value(s) corresponding to a data block to be transferred;

an application requesting the DMA engine to transfer a block of data;

the network interface storing one or more triggering value(s) corresponding to the data block to be transferred, along with an identification of the application which requested the DMA transfer;

the network interface monitoring the data stream being sent by the applications and comparing at least part of the data stream with the triggering value(s) stored in its memory; and

if any triggering value matches, indicating that that triggering value has matched.

73. A method according to claim 72, in which the application requests a DMA transfer by setting up a descriptor indicating the transfer required, and sending this descriptor to the DMA request queue address.

74. A method according to claim 72 or 73, in which after requesting a data transfer and storage of a triggering value, the application blocks until it receives a reschedule.

75. A method according to claim 72, 73 or 74, in which when a triggering value matches, a reschedule is sent to the application which requested the storage of that triggering value.

76. A method according to any of claims 72 to 75, in which, if the request queue is full when an application attempts to add a new request, the network interface indicates to that application that its requested transfer has failed.

77. A method according to any of claims 73 to 76, further comprising the steps of reading the first descriptor in the request queue and retrieving data from the main memory of the computer in accordance with the contents of the descriptor.

78. A method according to claim 77, further comprising the step of transmitting the data retrieved from the main memory in accordance with the content of the corresponding descriptor.

79. A method according to any of claims 72 to 78, further comprising the step of interrupting the transfer of a data block if the transfer is not completed after a predetermined length of time from the start of that transfer.

80. A method of transferring data from a sending application on a first computer to a receiving application on a second computer, each computer having a main memory, and a memory mapped network interface, the method comprising the steps of:

creating a buffer in the main memory of the second computer for storing data being transferred as well as data identifying one or more pointer memory location(s);

storing at said pointer memory location(s) at least one write pointer and at least one read pointer for indicating those areas of the buffer available for writes and for reads;

in dependence on the values of the WRP(s) and RDP(s), the sender application writing to the buffer;

updating the value of the WRP(s), after a write has taken place, to update the indication of the area(s) of the buffer available for reads and the area(s) available for writes;

in dependence on the values of WRP(s) and RDP(s), the receiver application reading from the buffer; and

updating the value of the RDP(s), after a read has taken place, to update the indication of the area(s) of the buffer available for reads and the areas(s) available for writes.

81. A method according to claim 80, in which the step of updating the value of the WRP(s) includes the sending application sending the updated value of the WRP to the main memory of the second computer, via the network.

82. A method according to claim 80 or 81, in which the first computer comprises a processing means with a cache memory, comprising the step of the sending application storing the value of the updated WRP in the cache memory.

83. A method according to claim 80, 81 or 82, in which the step of updating the value of the RDP(s) includes the receiving application sending the updated value of the RDP to the main memory of the first computer, via the network.

84. A method according to claim 80, 81, 82 or 83, in which the second computer comprises a processing means with a cache memory, the method comprising the step of the receiving application storing the value of the updated RDP in its cache memory.

85. A method according to any of claims 80 to 84, comprising the steps of:
the network interface of the second computer storing triggering value(s) corresponding to the address(es) of one or more write pointer(s) (WRP(s));
the network interface of the second computer monitoring the data stream received from the first computer and comparing at least part of the data stream with the triggering value(s) stored in its memory; and
if any triggering value matches, indicating that that triggering value has matched.

86. A method according to claim 85, in which when a triggering value is matched by the receipt of the WRP write instruction, a receiver interrupt is generated.

87. A method according to any of claims 80 to 86, further comprising the steps of:
providing a second buffer in the main memory of the second computer for storing write pointer data;
storing one or more second-buffer write pointer(s) and second-buffer read pointer(s) indicating the areas of the second-buffer available for writes and reads;
when the sending application writes to the first- buffer and updates the write pointer(s) of the first- buffer, writing to said second-buffer, in accordance with the value of the write pointer(s) and read pointer(s) of the second-buffer, the updated value of the write pointer of the first-buffer; and
updating the value of the second-buffer write pointer(s) to update the indication of the area(s) of the second-buffer available for writes and the areas(s) available for reads.

88. A method according to claim 87, further comprising the steps of:
reading a first-buffer write pointer value from the second buffer, in dependence on the contents of the second-buffer read pointer(s) and second-buffer write pointer(s),
and
reading from the first buffer in dependence on the value of a first-buffer pointer and the write pointer value read from the second buffer.
89. A method according to any of claims 80 to 88, further comprising the steps of:
the network interface of the first computer storing triggering value(s)
corresponding to address(es) of one or more RDP(s);
the network interface of the first computer monitoring the data stream received from the second computer and comparing at least part of the data stream with the triggering value(s) stored in its memory; and
if any triggering value matches, indicating that that triggering value has matched.
90. A method according to claim 89, in which when the network interface of the first computer matches a triggering value by the receipt of an RDP write instruction, a sender interrupt is generated.
91. A method according to any of claims 80 to 90, in which the sending application blocks if the values of the WRP(s) and RDP(s) indicate that the buffer is full.
92. A method according to claim 91, in which the sending application is unblocked on receipt of an interrupt.
93. A method according to any of claims 80 to 92, in which the receiving application blocks if the values of the WRP(s) and RDP(s) indicate that the buffer is empty.
94. A method according to claim 93, in which the receiving application is unblocked on receipt of an interrupt.

95. A method according to any of claims 80 to 94, in which a write pointer of a buffer points to the buffer address where the next byte of data should be written in that buffer.
96. A method according to any of claims 80 to 95, in which a read pointer of a buffer points to the buffer address of the first byte of data to be read from that buffer.
97. A method according to any of claims 80 to 96, in which when an application has written to the end of a buffer, it next writes to the start of the buffer, depending on the value of the WRP(s) and RDP(s) corresponding to that buffer.
98. A method according to any of claims 80 to 97, in which when an application has read to the end of a buffer, it next reads from the start of the buffer, depending on the value of the WRP(s) and RDP(s) corresponding to that buffer.
99. A method according to any of claims 80 to 98, in which the value of one or more WRP(s) and/or RDP(s) is updated when a triggering value is matched in a network interface.
100. A computer network comprising two computers, the first computer running a sending application and the second computer running a receiving application, each computer having a main memory and a memory mapped network interface, the main memory of the second computer having: a buffer for storing data being transferred between computers as well as data identifying one or more pointer memory location(s);
means for reading at least one write pointer (WRP) and at least one read pointer (RDP) stored at (a) pointer memory location(s), for indicating the areas of the buffer available for writes and the area(s) available for reads;
the network interface of the second computer comprising:
a memory mapping;
means for reading data from the buffer in accordance with the contents of the WRP(s) and RDP(s); and

means for updating the value of the RDP(s) after a read has taken place, to update the indication of the area(s) of the buffer available for reads and the area(s) available for writes.

101. A computer network according to claim 100, the network interface of the first computer comprising:

a mapping memory; and

means for sending data to the buffer of the second computer.

102. A computer network according to claim 100 or 101, the main memory of the second computer storing the value of at least one WRP.

103. A computer network according to claim 100, 101 or 102, in which one or more pointer memory location(s) are in the main memory of the first computer.

104. A computer network according to any of claims 100 to 103, in which one or more pointer memory location(s) are located in the main memory of the second computer.

105. A computer network according to any of claims 100 to 104, in which the first computer comprises a processing means with a cache memory, with one or more WRP(s) and/or RDP(s) stored in that cache memory.

106. A computer network according to any of claims 100 to 105, in which the second computer has a processing means with a cache memory, with one or more WRP(s) and/or RDP(s) stored in that cache memory.

107. A computer network according to any of claims 100 to 106, in which the network interface of the first computer comprises:

means for writing data to the buffer in accordance with the values of at least one RDP and one WRP, using its memory mapping; and

means for updating the value of the WRP(s) to update the indication of the area(s) of the buffer available for reads and the area(s) available for writes.

108. A computer network according to any of claims 100 to 107, in which the main memory of the second computer comprises a second buffer; and the computer network also having:

means for reading one or more write pointer(s) and one or more read pointer(s) of the second buffer indicating the areas of the second buffer available for writes and those available for reads;

means for updating the write pointer(s) of the first buffer, when an application running on one of the computers writes to the first buffer;

means for writing to said second buffer, in accordance with the value of the write pointer(s) and read pointer(s) of the second buffer, the updated value of the write pointer of the first buffer; and

means for updating the value of the second buffer's write pointer(s) to update the indication of the area(s) of the second buffer available for reads and the area(s) available for writes.

109. A computer network according to claim 108, further comprising means for storing one or more write pointer(s) of the second buffer indicating the areas of the second buffer available for reads and the area(s) available for writes.

110. A computer network according to any of claims 80 to 109, in which the first and/or second buffer is a circular buffer.

111. A computer network according to claim 108, 109 or 110, in which the network interface of the second computer also comprises:

means for reading a first-buffer WRP value from the second buffer in accordance with the values of the second-buffer WRP(s) and RDP(s);

means for updating the RDP(s) of the second buffer to update the indication of the areas of the second buffer available for reads and writes;

means for reading from the first buffer in accordance with the contents of the first-buffer WRP value read from the second buffer, and a first-buffer RDP; and

means for updating the value of the RDP(s) of the first buffer to update the indication of the area(s) of the first buffer available for reads and writes when an application running on the second computer reads from the first buffer.

112. A computer network according to any of claims 80 to 111, the network interface of one or both computers also comprising:

fa memory for storing triggering value(s), corresponding to one or more address(es) of WRP(s) and/or RDP(s);

means for monitoring a data stream being transferred between the two computers and for comparing at least part of the data stream being transferred with the stored triggering value(s); and

means for indicating that a triggering value has been matched, when the part of the data stream being compared matches a triggering value.

113. A computer network according to claim 112, in which the means for indicating that a triggering value has been matched comprises means for generating an interrupt.

114. A method of sending a request from a client application on a first computer to a server application on a second computer, and sending a response from the server application to the client application, both computers having a main memory and a memory mapped network interface, the method comprising the steps of:

(A) providing a buffer in the main memory of each computer;

(B) the client application, providing software stubs which produce a marshalled stream of data representing the request;

(C) the client application sending the marshalled stream of data to the server's buffer;

(D) the server application unmarshalling the stream of data by providing software stubs which convert the marshalled stream of data into a representation of the request in the server's main memory;

(E) the server application processing the request and generating a response;

(F) the server application providing software stubs which produce a marshalled stream of data representing the response;

(G) the server application sending the marshalled stream of data to the client's buffer; and

(H) the client application unmarshalling the received stream of data by providing software stubs which convert the received marshalled stream of data into a representation of the response in the client's main memory.

115. A method according to claim 114 in which in step (c) and/or step (g) the stream of marshalled data is sent according to the method of any of claims 80 to 99.

116. A method according to claim 114 or 115, comprising the step of the client and server stubs sending the marshalled streams of data directly over the network, using the memory mapped network interfaces.

117. A method according to claim 114, 115 or 116, in which the sending and/or marshalling of a response by the server application may take place at the same time as the client application is unmarshalling the response from its buffer.

118. A method according to any of claims 114 to 117, in which the sending and/or marshalling of a request by the client application may take place at the same time as the server application is unmarshalling the request from its buffer.

119. A method according to any of claims 114 to 118, in which the response generated by the server application comprises two or more parts;

the server application providing software stubs which convert at least a first part of the response into a marshalled stream of data;

the server application sending the marshalled data stream representing the first part of the response to the client's buffer;

one or more parts of the response being provided by a hardware device in the server computer in the form of a marshalled stream of data; and

the hardware device sending its marshalled stream of data to the client's buffer.

120. A method according to claim 119, in which one or more parts of the response generated by the server application is provided by another software application running on the second computer in the form of a marshalled stream of data; and
the software application sending its marshalled stream of data to the client's buffer.

121. A method according to claim 119 or 120, in which each part of the response is sent to an appropriate part of the client's buffer such that when all parts of the response have been received in the buffer, the contents of the buffer comprise a marshalled data stream representing the whole response from the server application.

122. A method according to any of claims 114 to 121, comprising the steps of:
the network interface of the first computer storing triggering value(s)
corresponding to a property of one or more parts of the expected response;
the network interface of the first computer monitoring the response received from the server application and comparing at least part of the data stream with the triggering value(s) stored in its memory; and
if any triggering value matches, indicating that that triggering value has matched.

123. A method according to claim 122, comprising the step of sending an interrupt when a triggering value matches.

124. A method according to claim 122 or 123, comprising the step of changing the value of a counter when a triggering value is matched.

125. A method according to claims 122, 123 or 124, in which the client application, while it is waiting for the response from the server application, blocks or polls an event counter.

126. A method of arranging data for transfer as a data burst over a computer network comprising the steps of: providing a header comprising the destination address of a

certain data word in the data burst, and a signal at the beginning or end of the data burst for indicating the start or end of the burst, the destination addresses of other words in the data burst being inferable from the address in the header.

127. A method according to claim 126, in which the signal identifying the end of a burst comprises a null signal.

128. A method of processing a data burst received over a computer network comprising the steps of:

reading a reference address from the header of the data burst, and

calculating the addresses of each data word in the burst from the position of that data word in the burst in relation to the position of the data word to which the address in the header corresponds, and from the reference address read from the header.

129. A method of interrupting transfer of a data burst over a computer network comprising the steps of:

halting transfer of a portion of the data burst which has not yet been transferred, thereby splitting the data burst into two burst sections, one which is transferred, and one waiting to be transferred.

130. A method of restarting transfer of a data burst that has been interrupted according to the method of claim 129, comprising the steps of:

calculating a new reference address for the untransferred data burst section from the address contained in the header of the whole data burst, and from the position in the whole data burst of the first data word of the untransferred data burst section in relation to the position of the data word to which the address in the header corresponds;

providing a new header for the untransferred data burst section comprising the new reference address; and

transmitting the new header along with the untransferred data burst section.

131. A method according to claim 130, comprising calculating the new reference address for the untransferred data burst section from the reference address contained in

the header of the whole data burst and from the number of data words in the transferred data burst section.

132. An apparatus for synchronising an end-point application in a computer, comprising:

means for generating and storing at least one code whose purpose is to associate an action with an address-based event in an information stream, which comprises data and associated memory addresses, on an information pathway within the computer;

means for comparing the generated code with each of at least the associated addresses to detect the address-based event; and

means for performing the associated action in response to detection of the address-based event.

133. A method of transferring data to a buffer of a receiving computer application, comprising the steps of:

storing in a sending application a write pointer representing the position of the start of a first section of the buffer available for receiving data;

storing in the receiving application a copy of the write pointer;

sending a first block of data to the first section; and

updating the write pointer in the sending application and the copy of the write pointer in the receiving application.

134. A method as claimed in claim 133, comprising the steps of:

storing in the receiving application a read pointer representing the position of the start of a second section of the buffer containing data available for reading;

storing in the sending application a copy of the read pointer;

reading in the receiving application a second block of data from the second section; and

updating the read pointer in the receiving application and the copy of the read pointer in the sending application.

135. A method as claimed in claim 134, comprising the steps of:

comparing in the sending application the write pointer and the copy of the read pointer to determine the size of the first section; and

sending the first block of data which is no bigger than the size of the first section.

136. A method as claimed in claim 134 or 135, comprising the steps of:

comparing in the receiving application the read pointer and the copy of the write pointer to determine the size of the second section; and

reading in the receiving application from the second section the second block of data which is no bigger than the size of the second section.

137. A method as claimed in any one of claims 133 to 136, in which the first block of data is sent from the sending computer application.

138. A method as claimed in any one of claims 133 to 137, in which the buffer is a circular buffer.

139. A method as claimed in any one of claims 133 to 138, in which the first application is on a first computer and the second application is on a second computer separated from the first computer by a network.

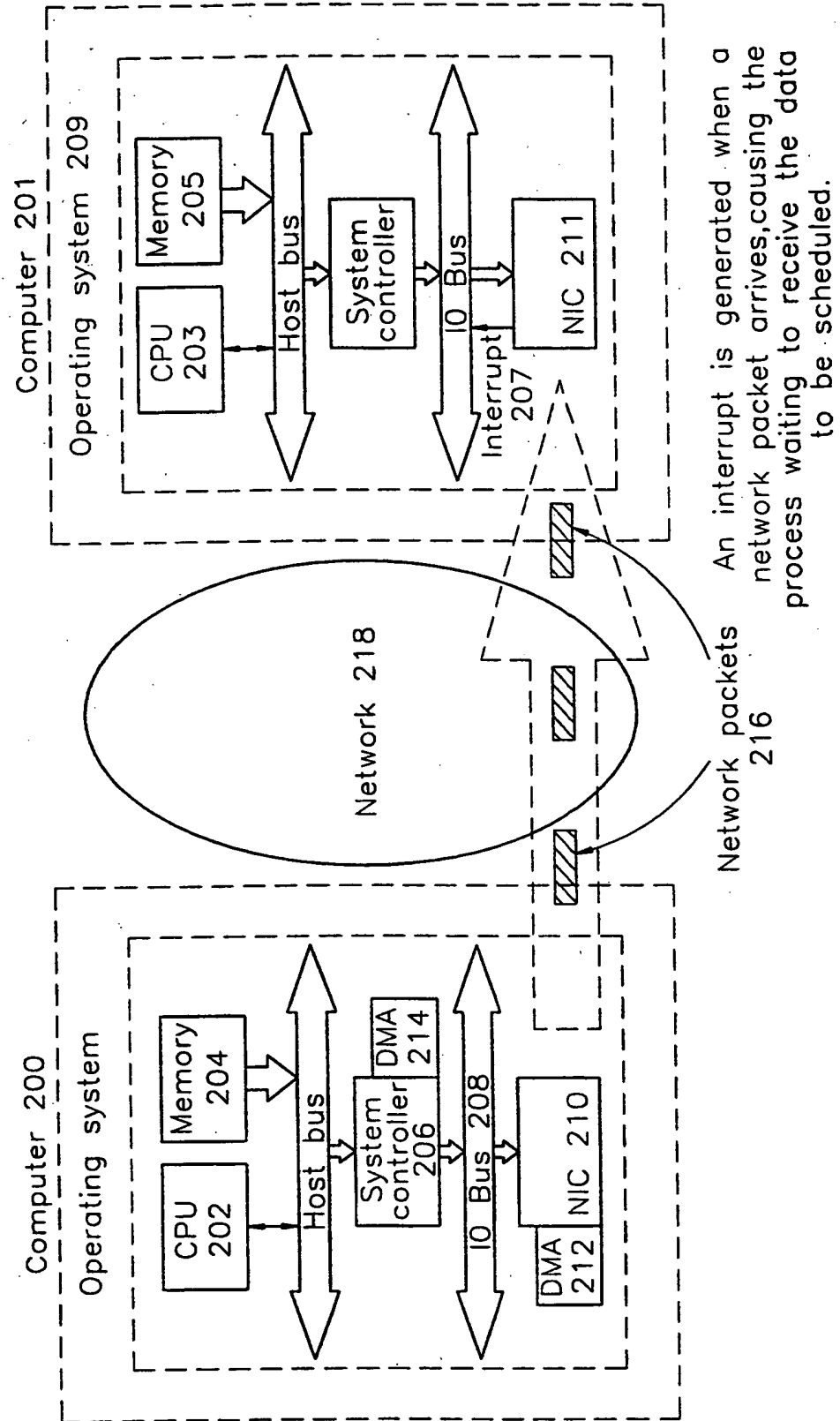


FIG 1

Prior art.

Synchronous computer networks

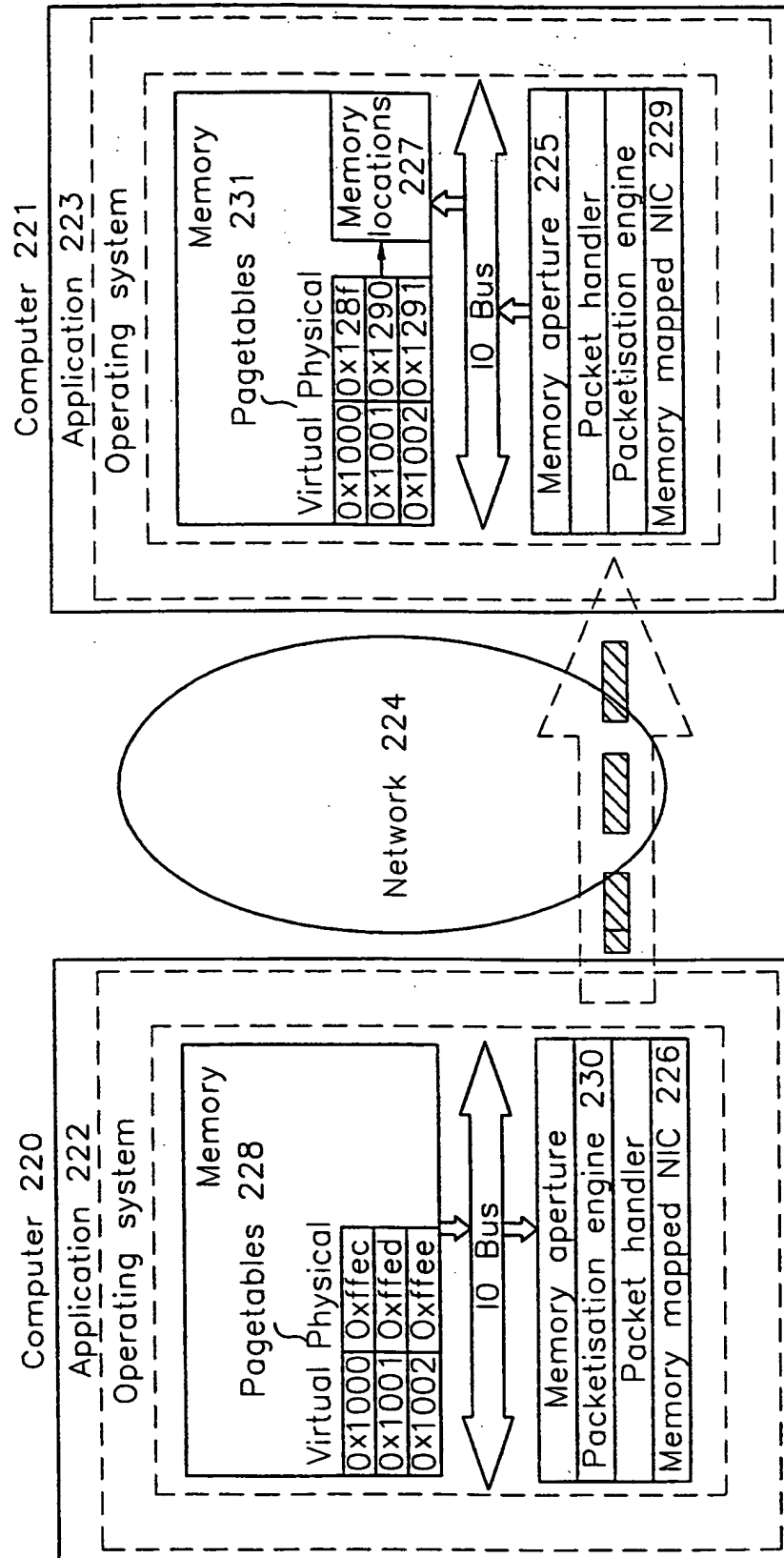


FIG 2

Prior art.

Memory mapped networking

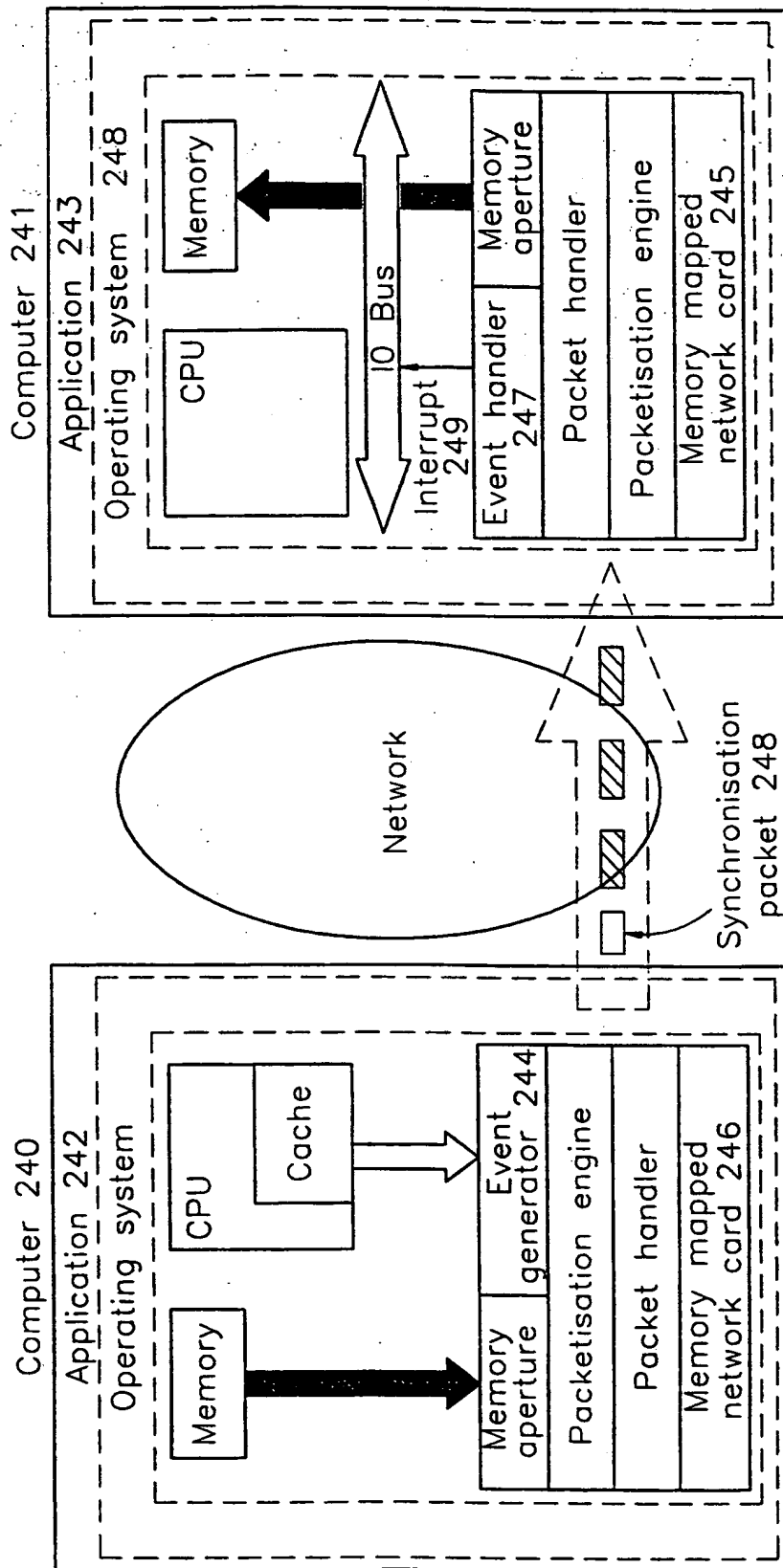


FIG 3
Prior art.
Showing synchronisation in a
memory mapped network

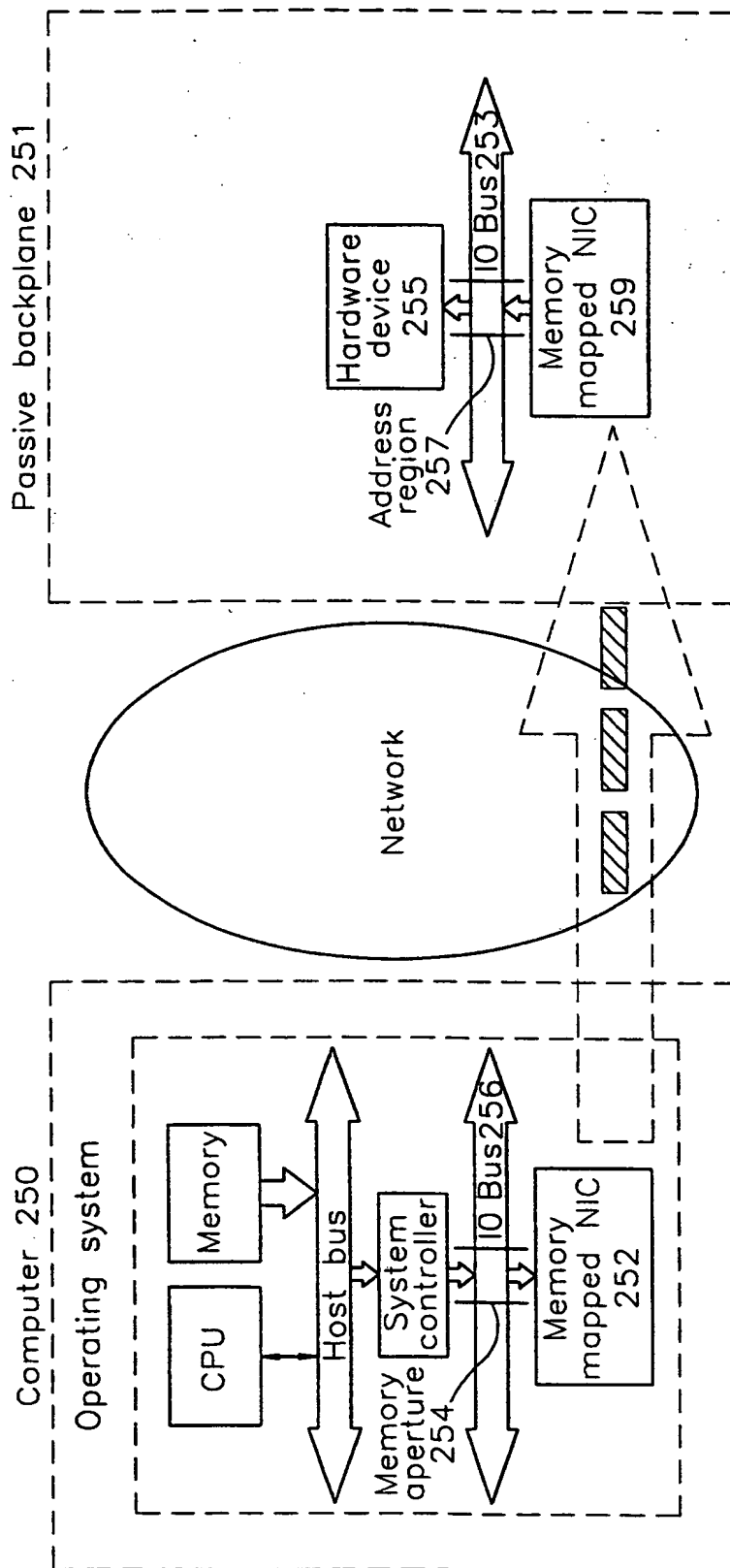


FIG 4
Prior art.
Hardware communication over
a memory mapped network.

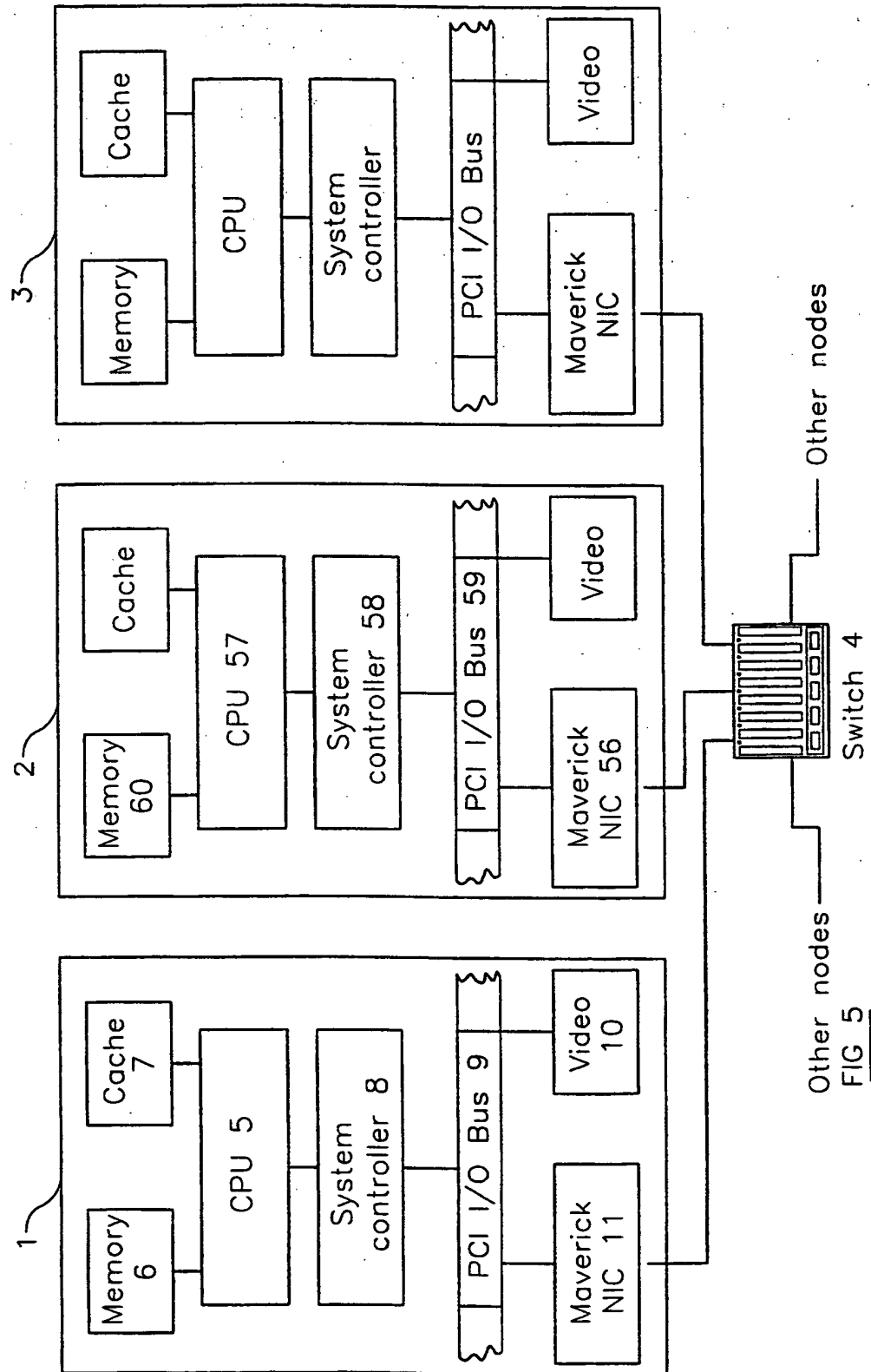


FIG 5
Showing NICs of present invention being used to construct a computer network

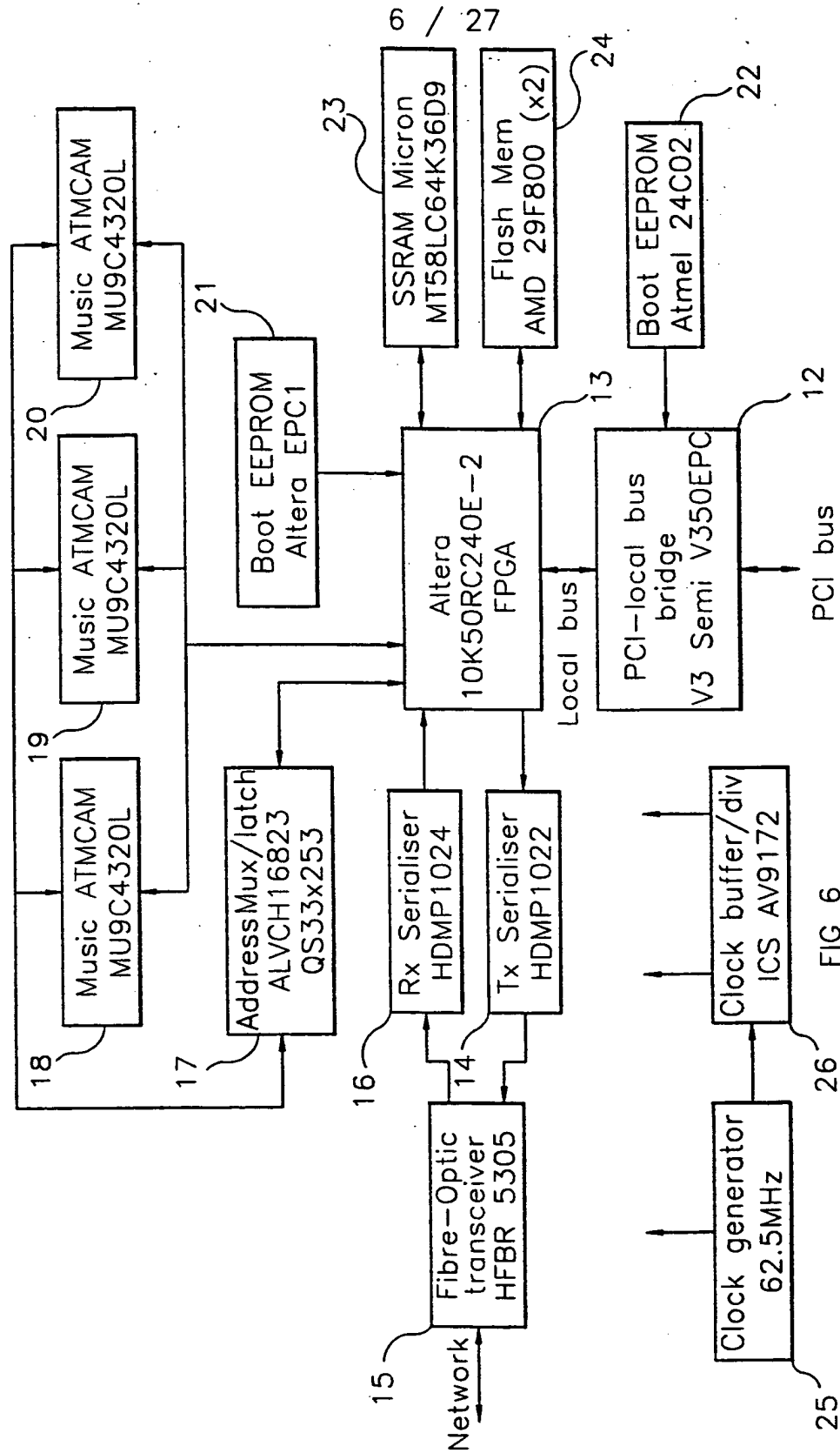


FIG 6

Showing the functional blocks of the
NIC in one embodiment of the invention

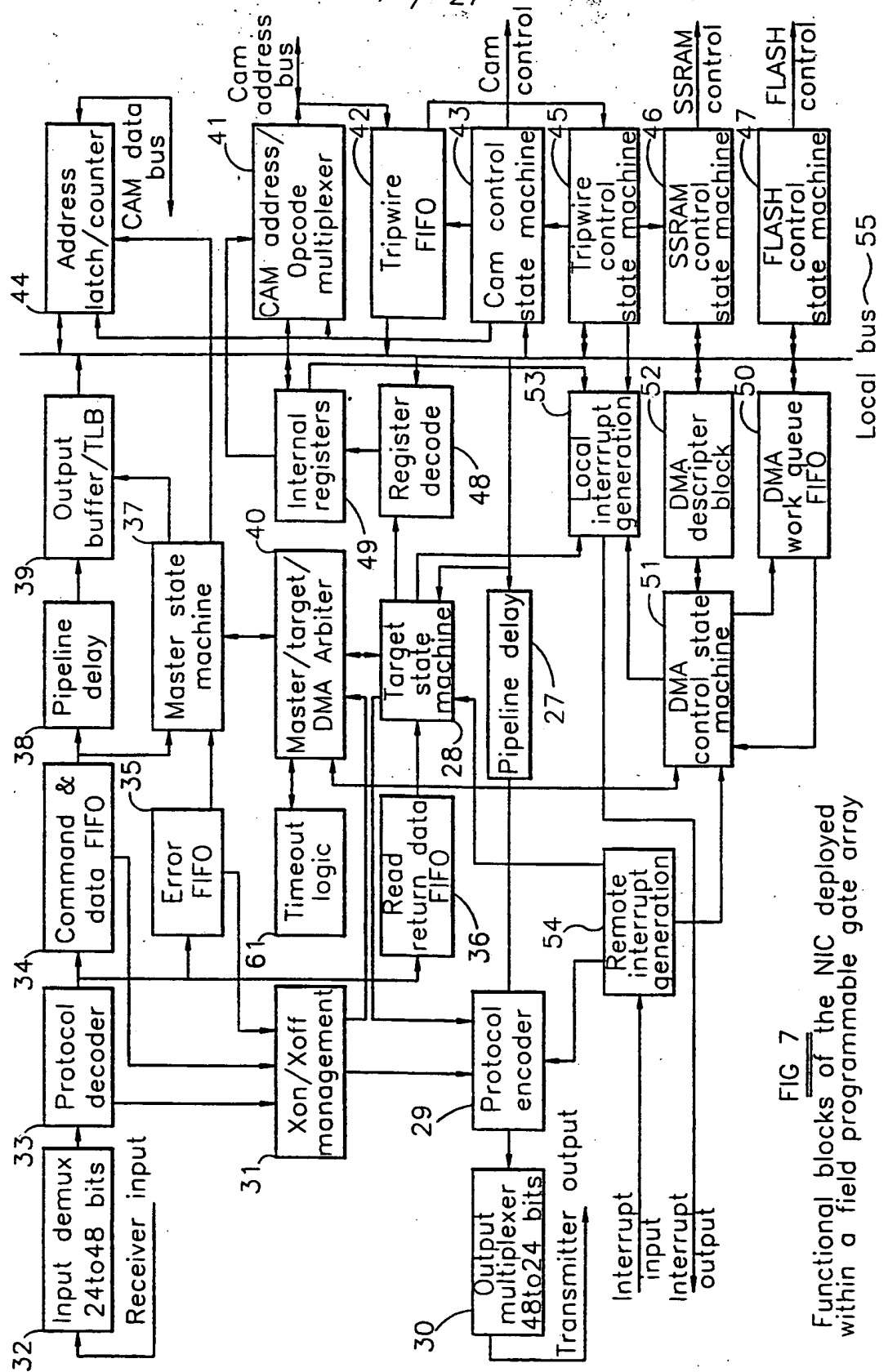
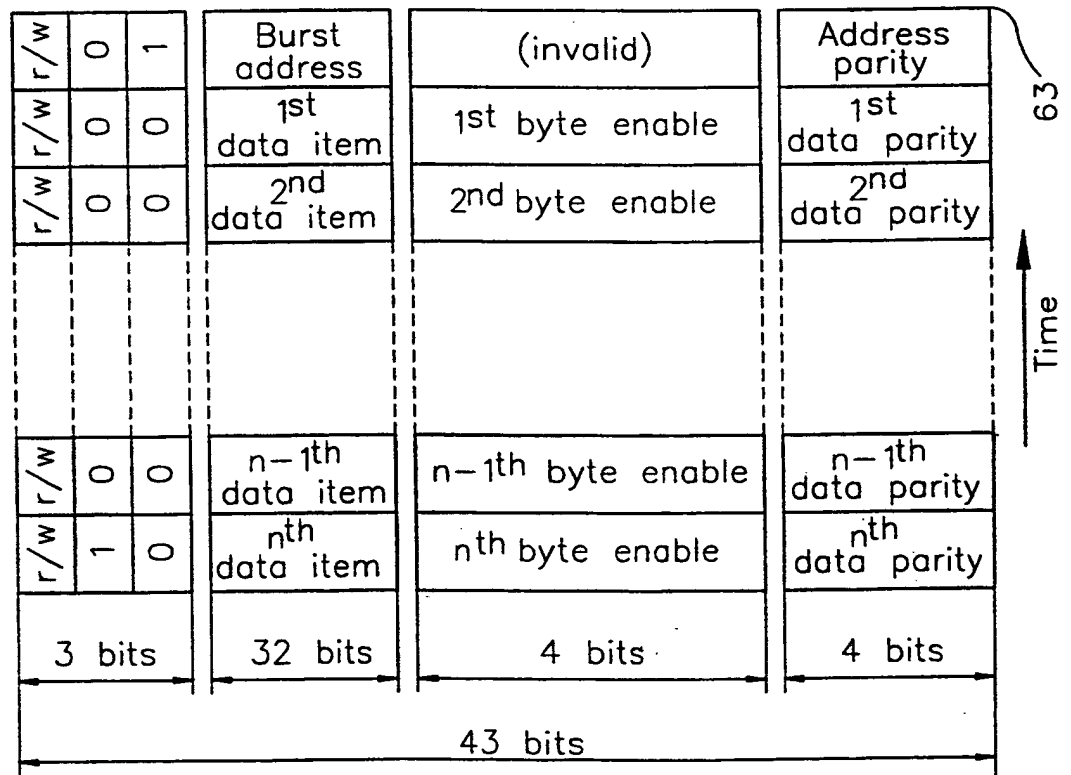


FIG 7

Functional blocks of the NIC deployed within a field programmable gate array

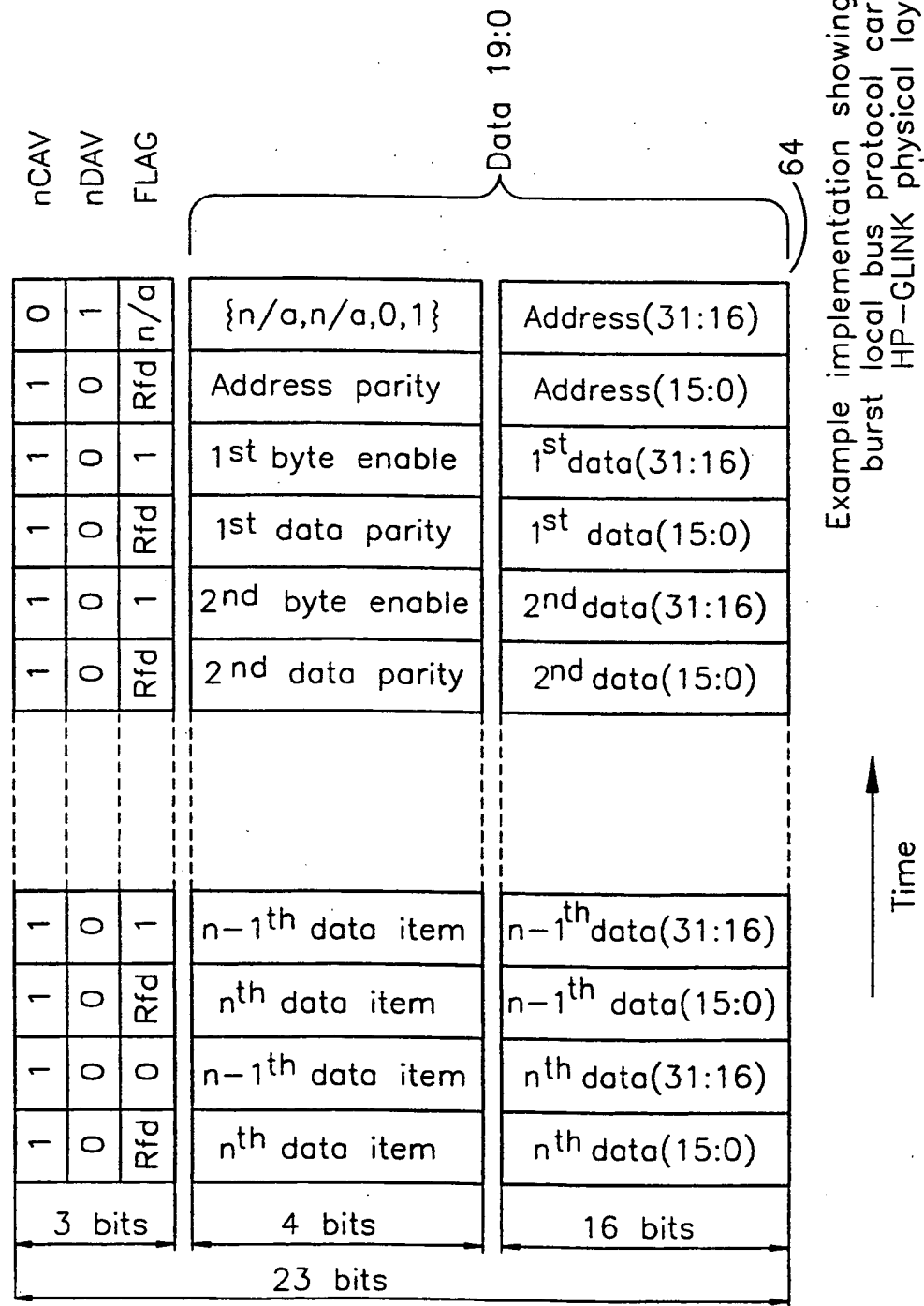
Read/Write
Burst last (BLAST)
Address (ADS)



Example implementation showing i960-style
burst local bus protocol carried on HP-GLINK
physical layer

FIG 8a

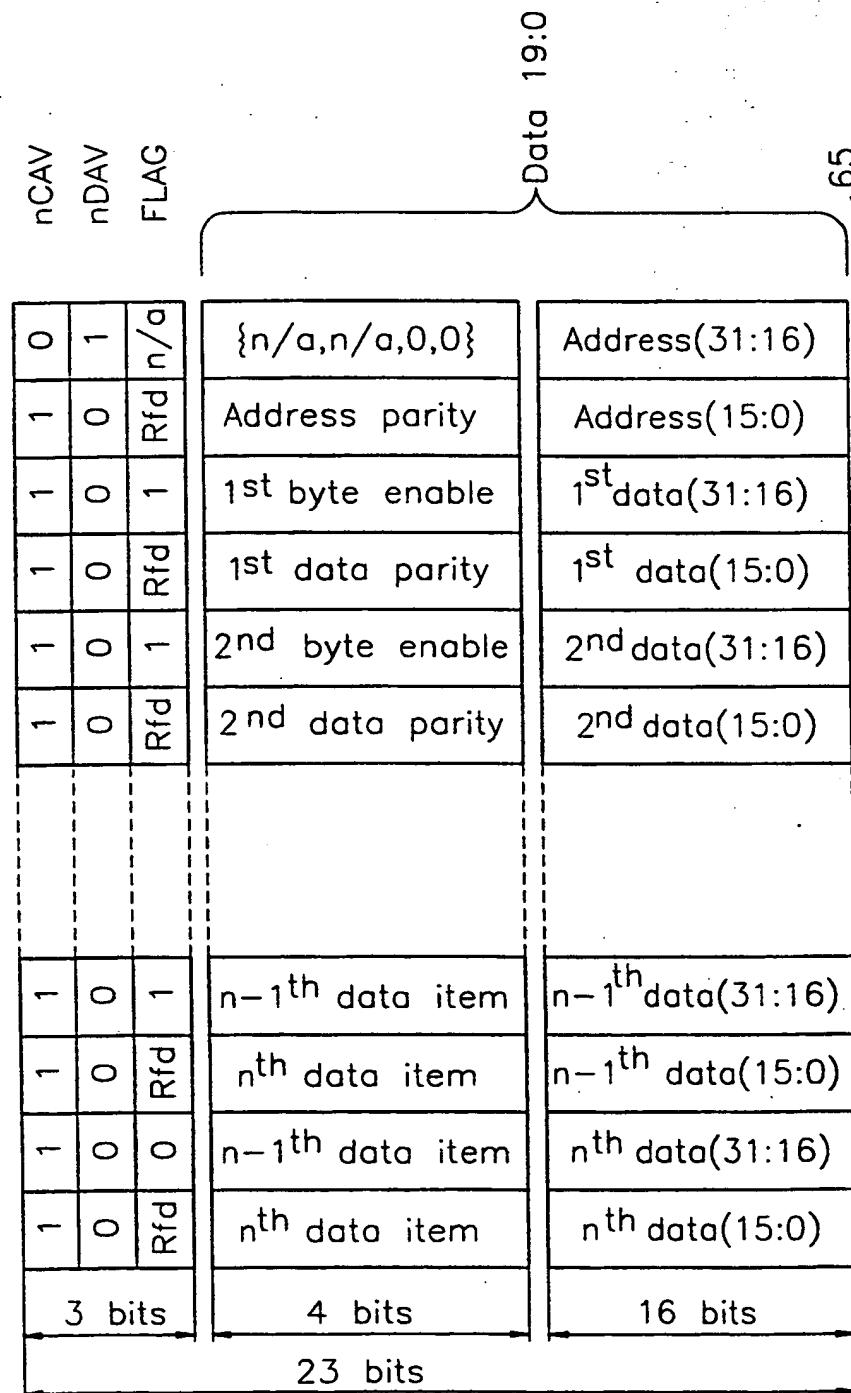
Current protocol embodiment



Example implementation showing i960-style
burst local bus protocol carried on
HP-GLINK physical layer

FIG 8b

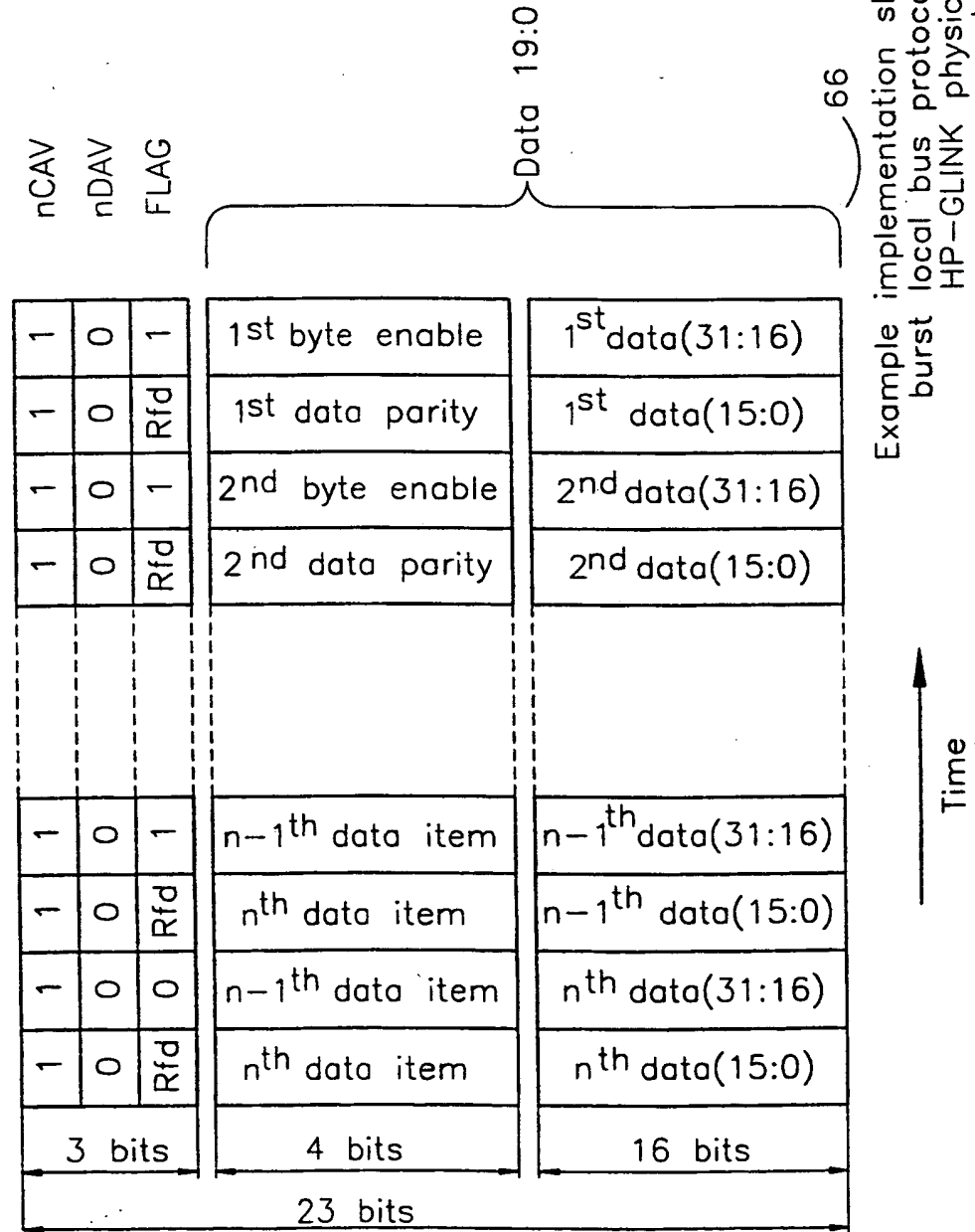
Current protocol embodiment (Write burst)



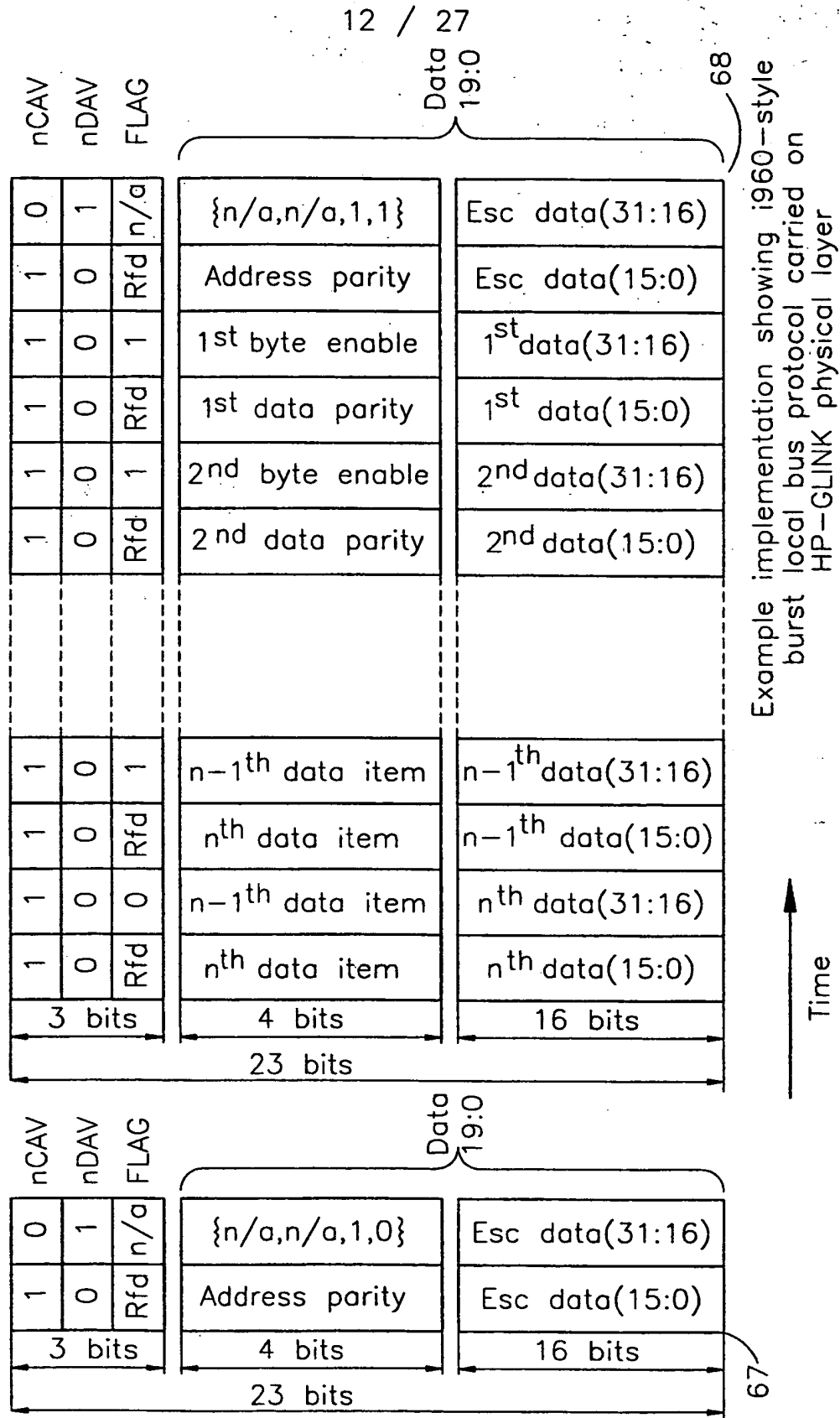
Example implementation showing i960-style
burst local bus protocol carried on
HP-GLINK physical layer

FIG 8c

Current protocol embodiment(Read burst)

FIG 8^d

Current protocol embodiment(Read data burst returning)



Example implementation showing i960-style burst local bus protocol carried on HP-GLINK physical layer

FIG 8^e

Current protocol embodiment(Escape packets without / with payloads)

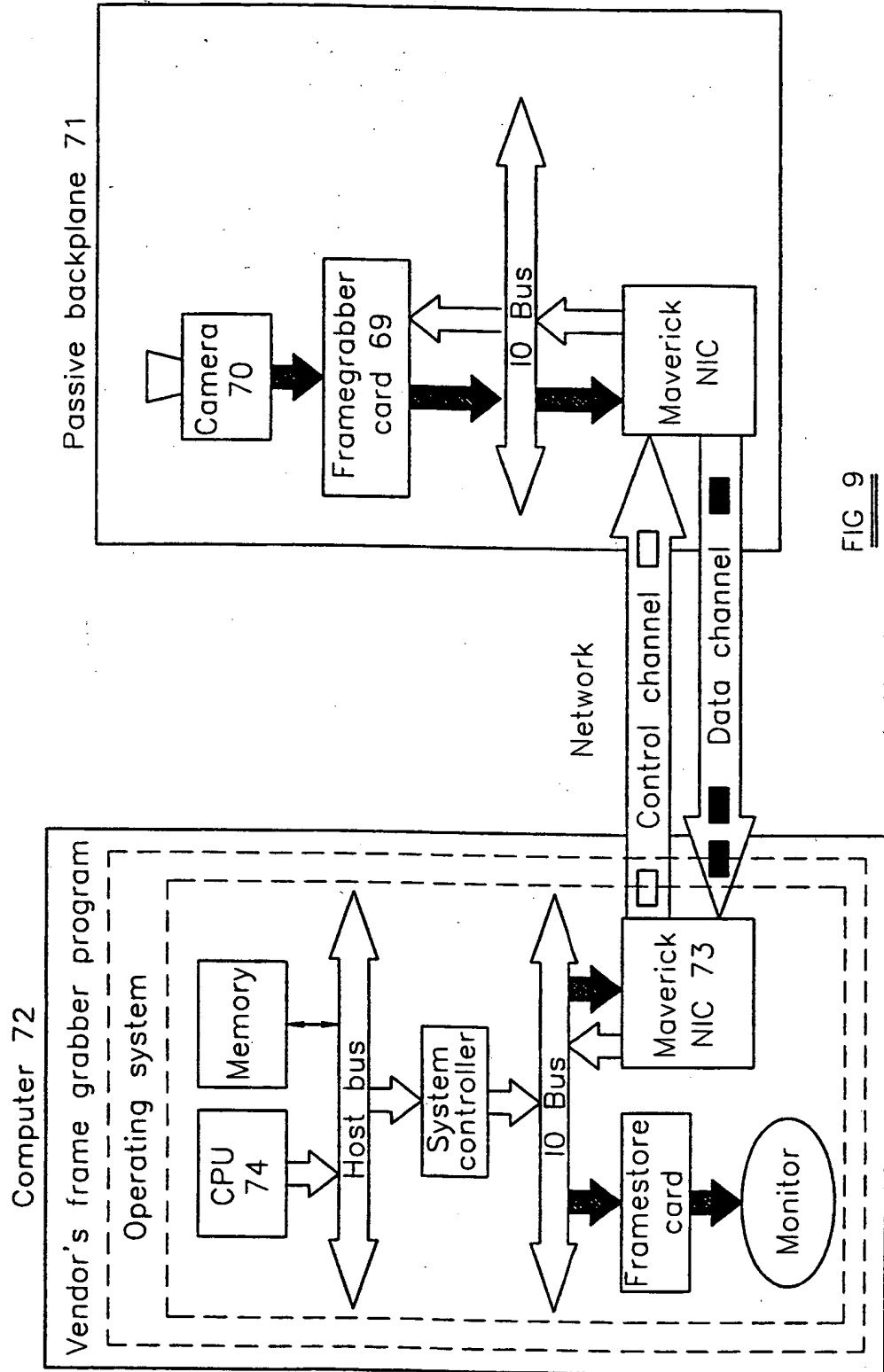


FIG 9

Hardware communication
using the NIC of FIG 6.

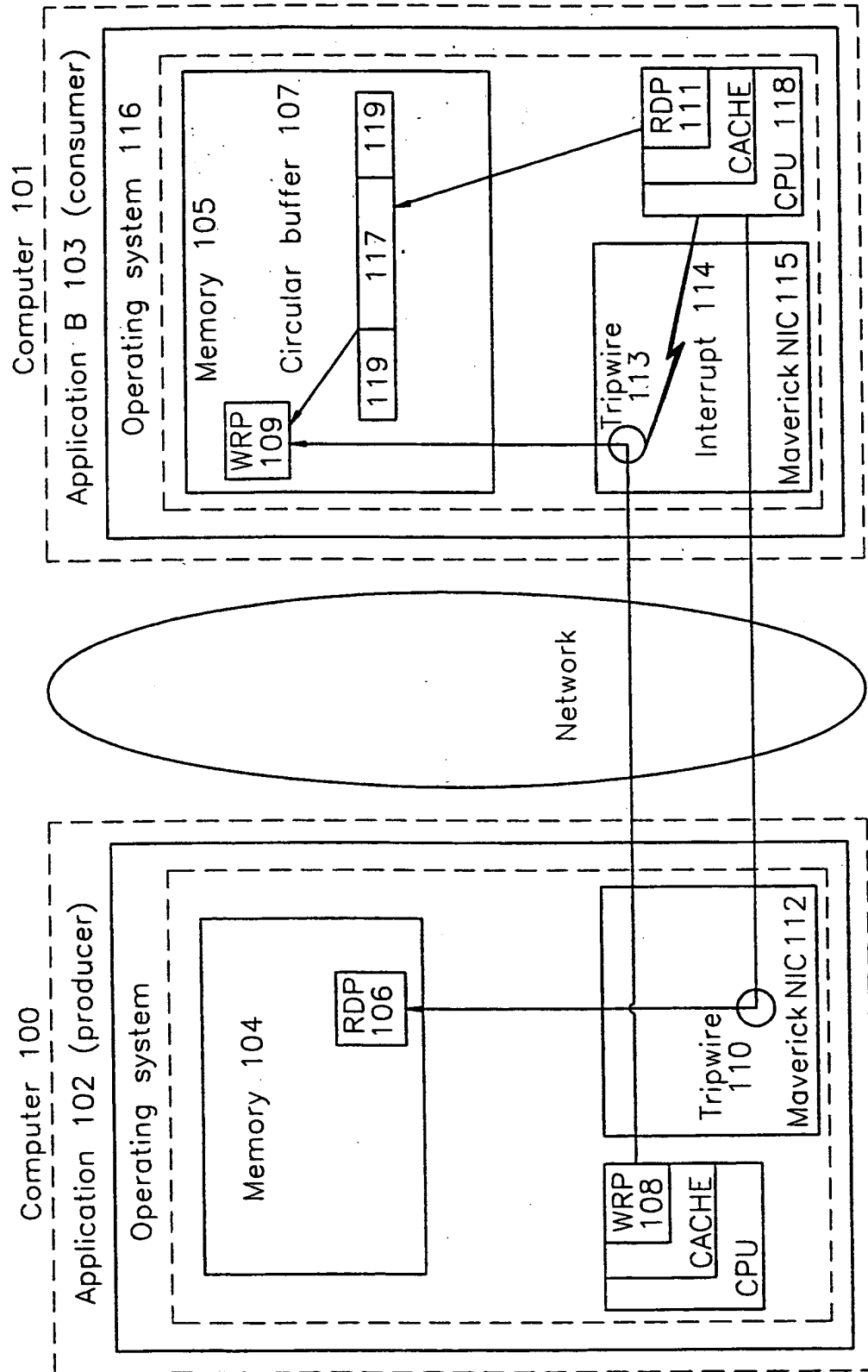


FIG 10
Circular buffer support using the NIC of FIG 6.

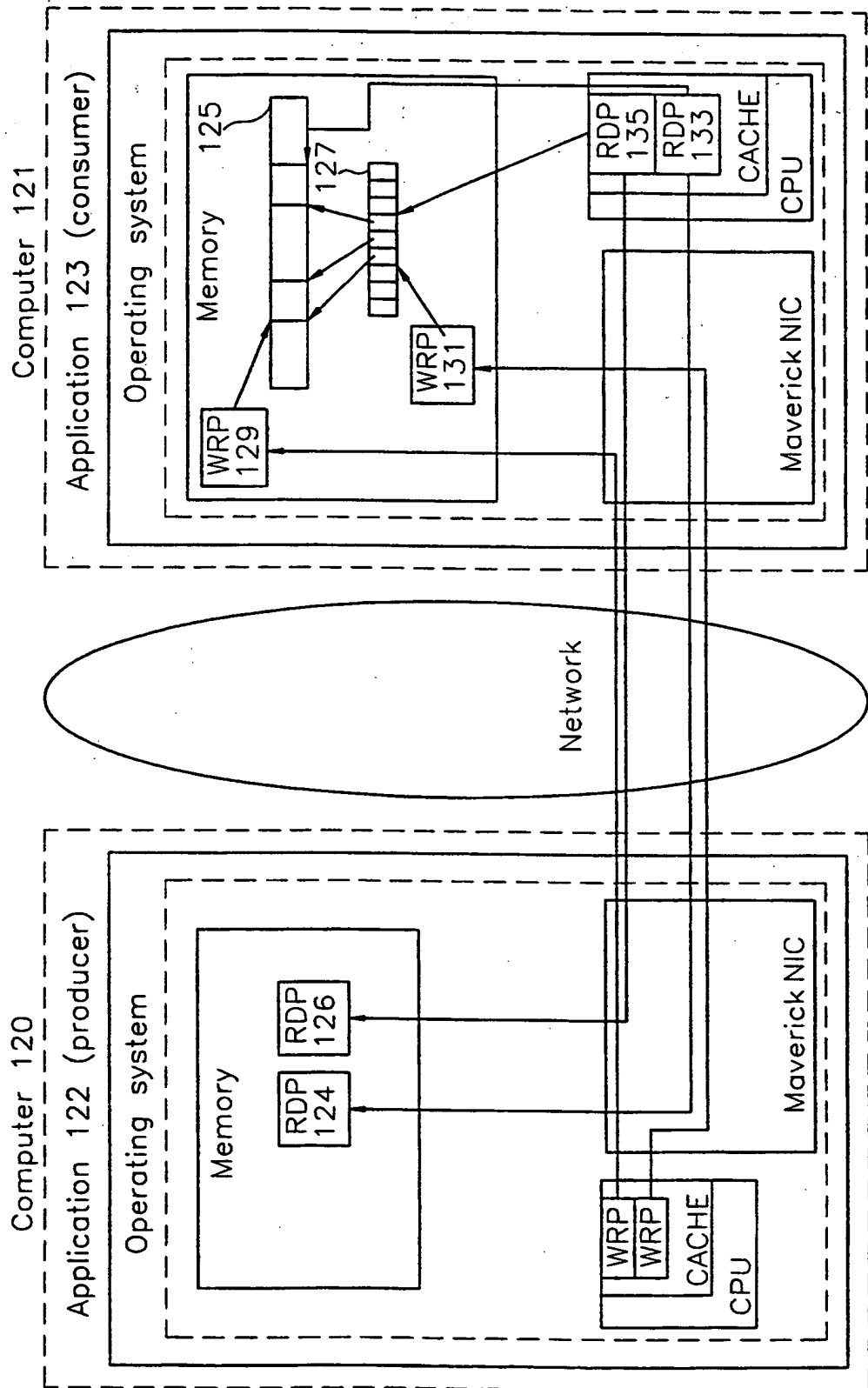


FIG 11

Support for discrete message communication using circular buffers

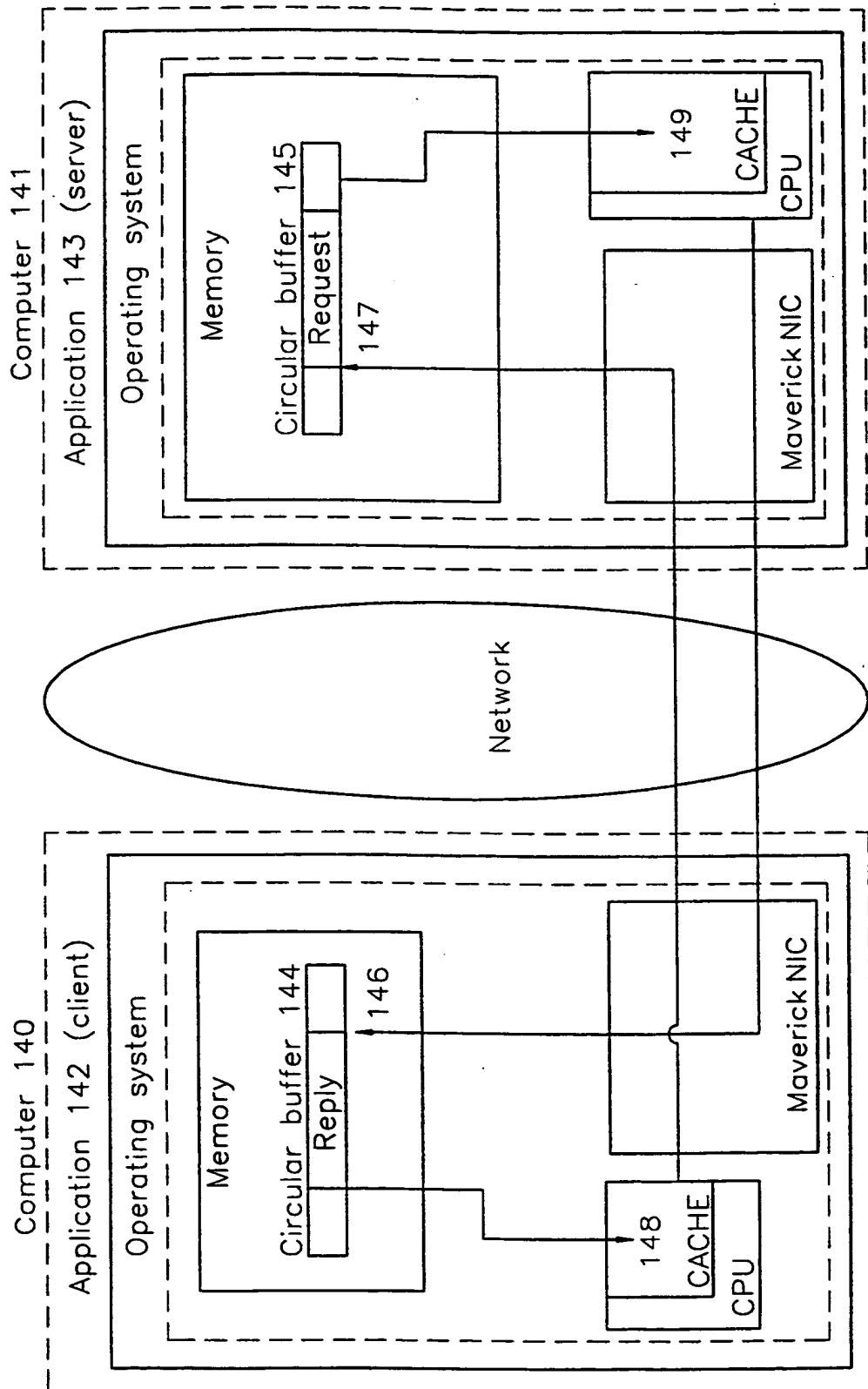


FIG 12

Client-server interaction using two NICs

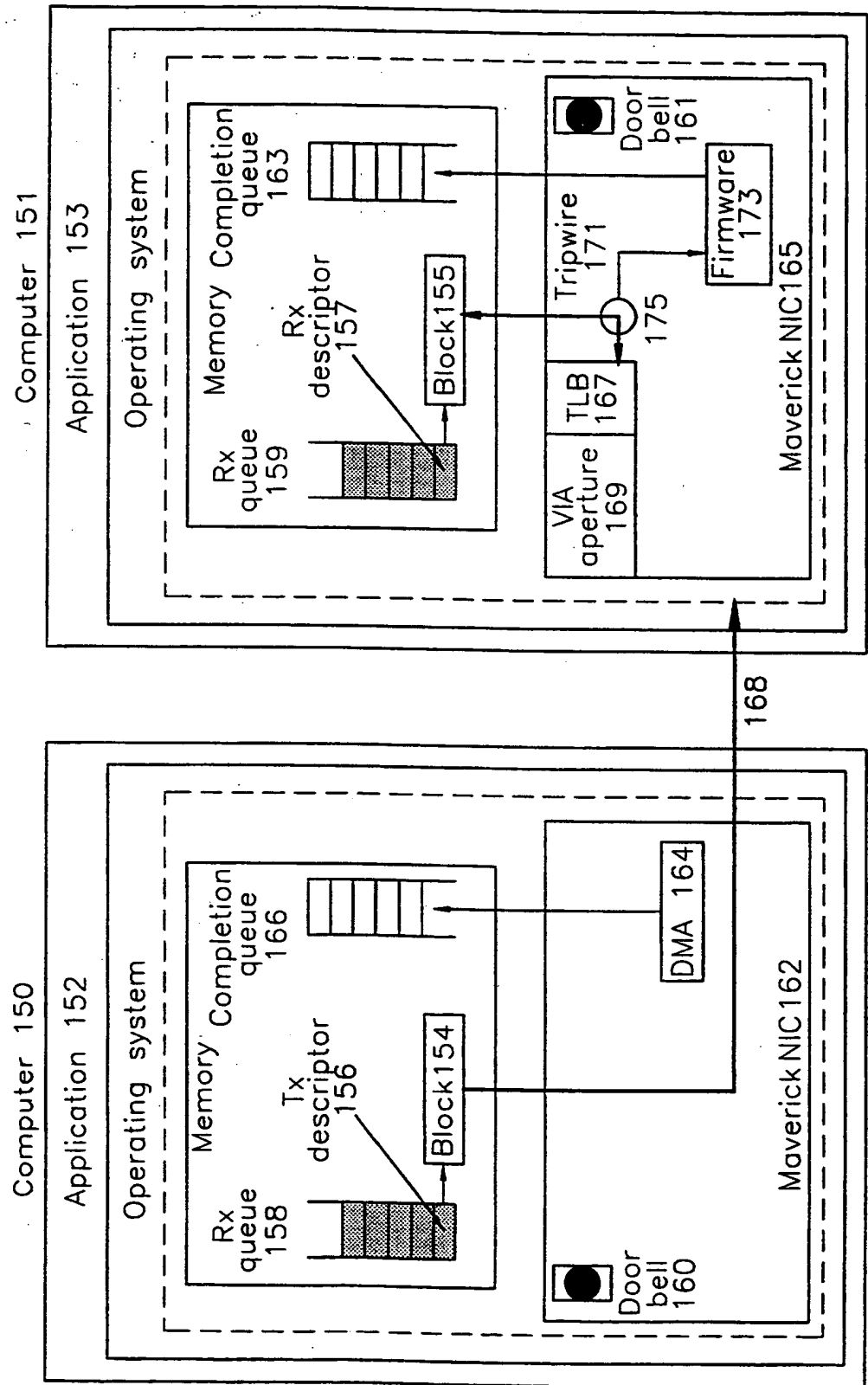


FIG 13

VIA support

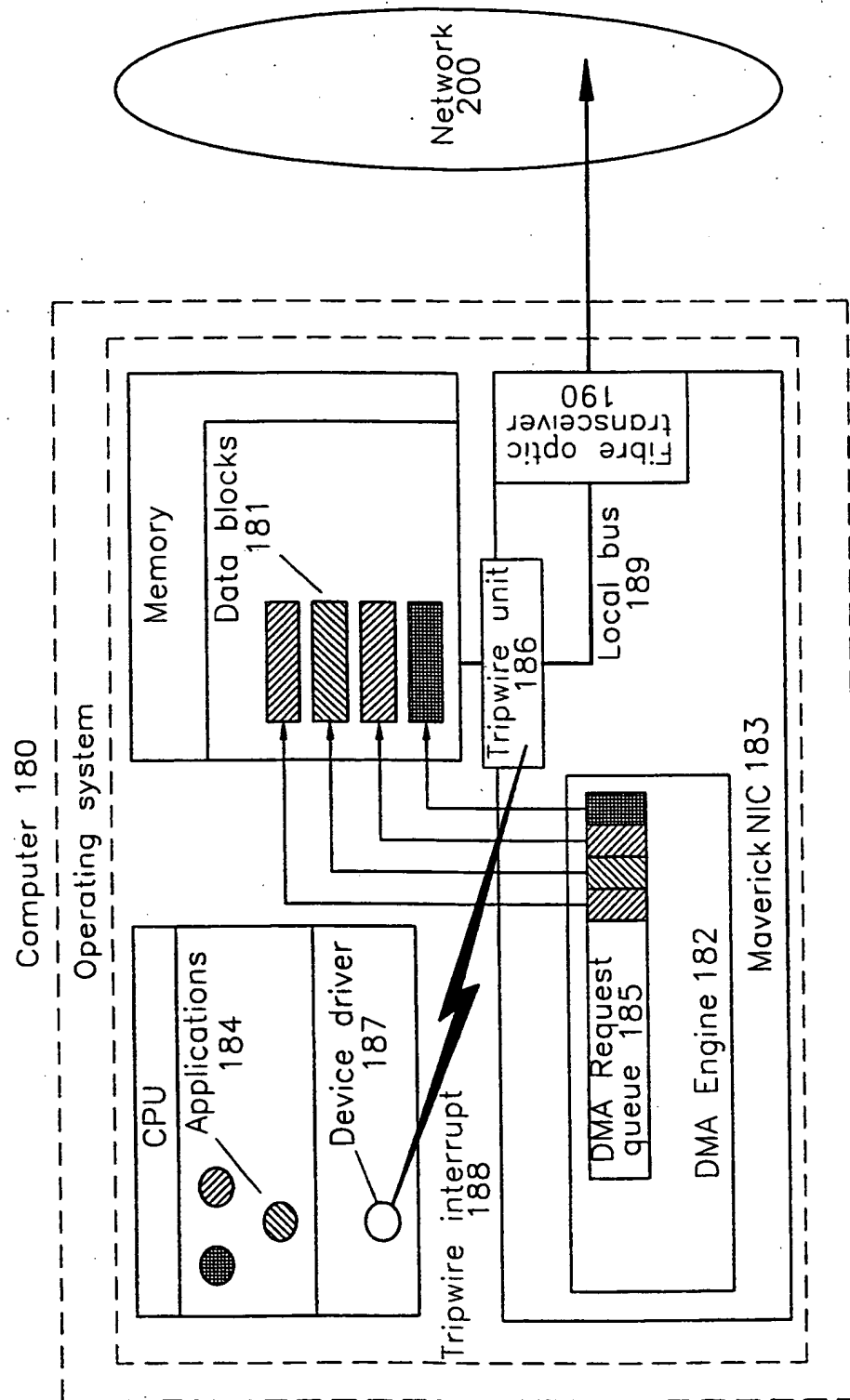


FIG 14
Outgoing stream synchronisation

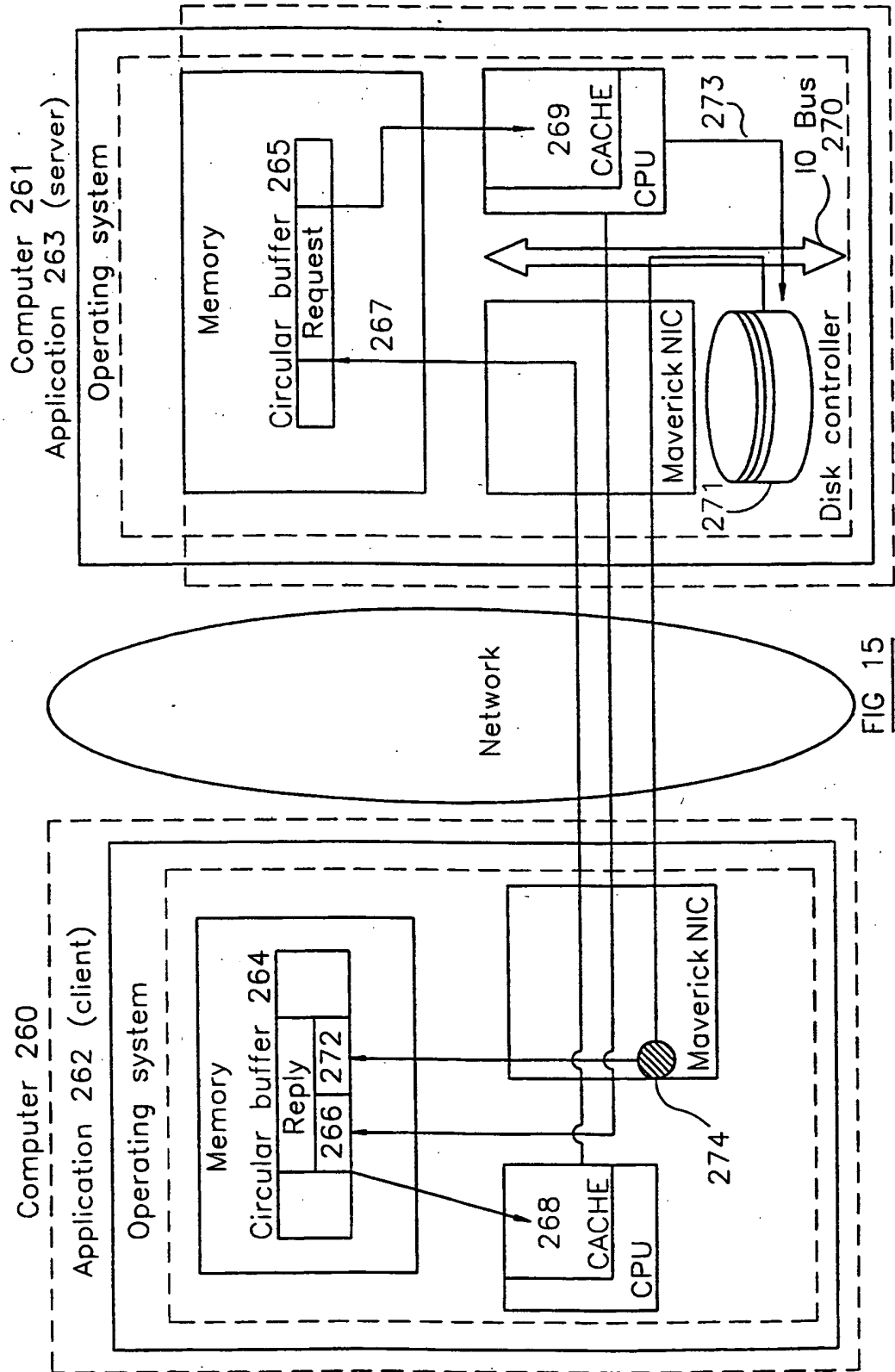
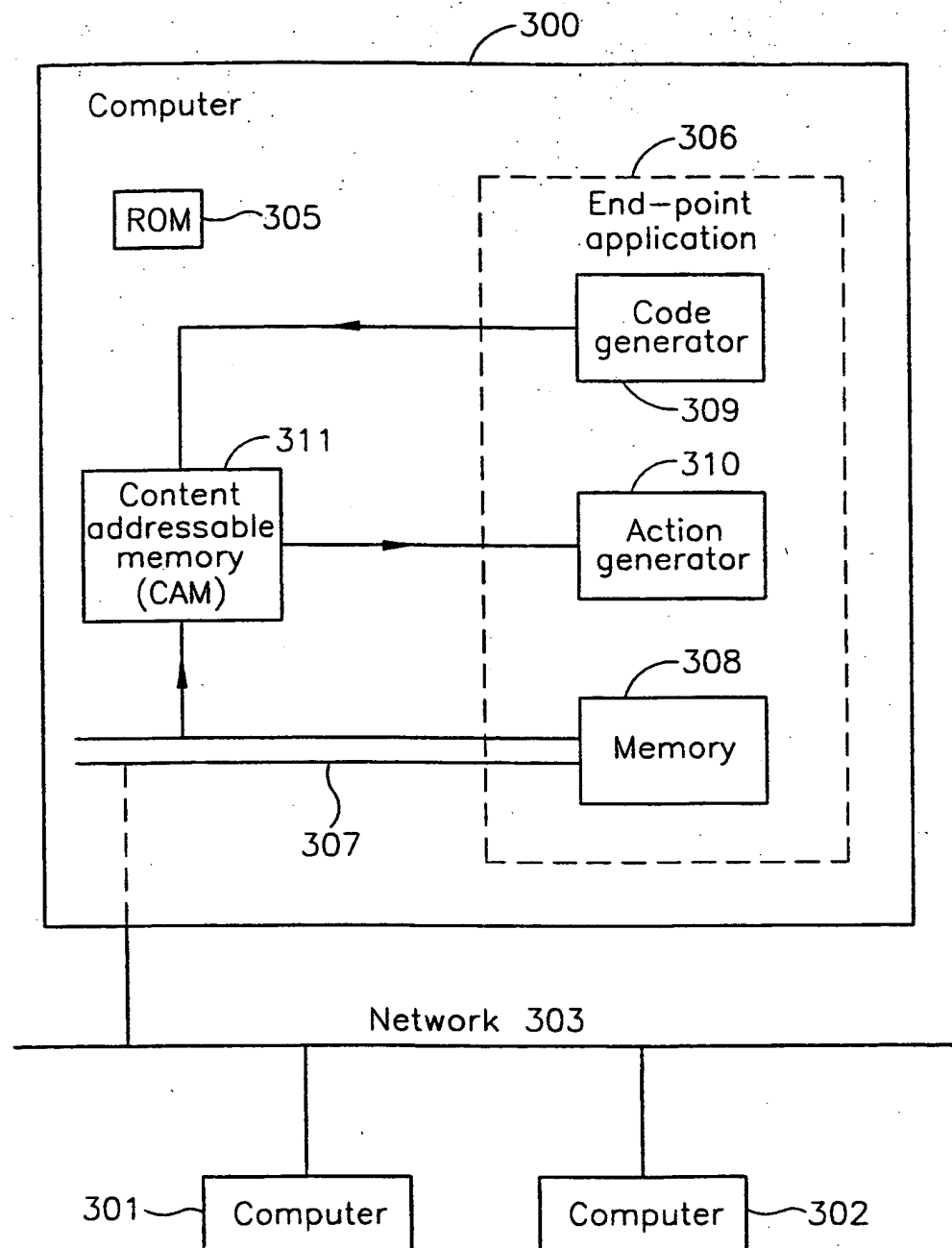
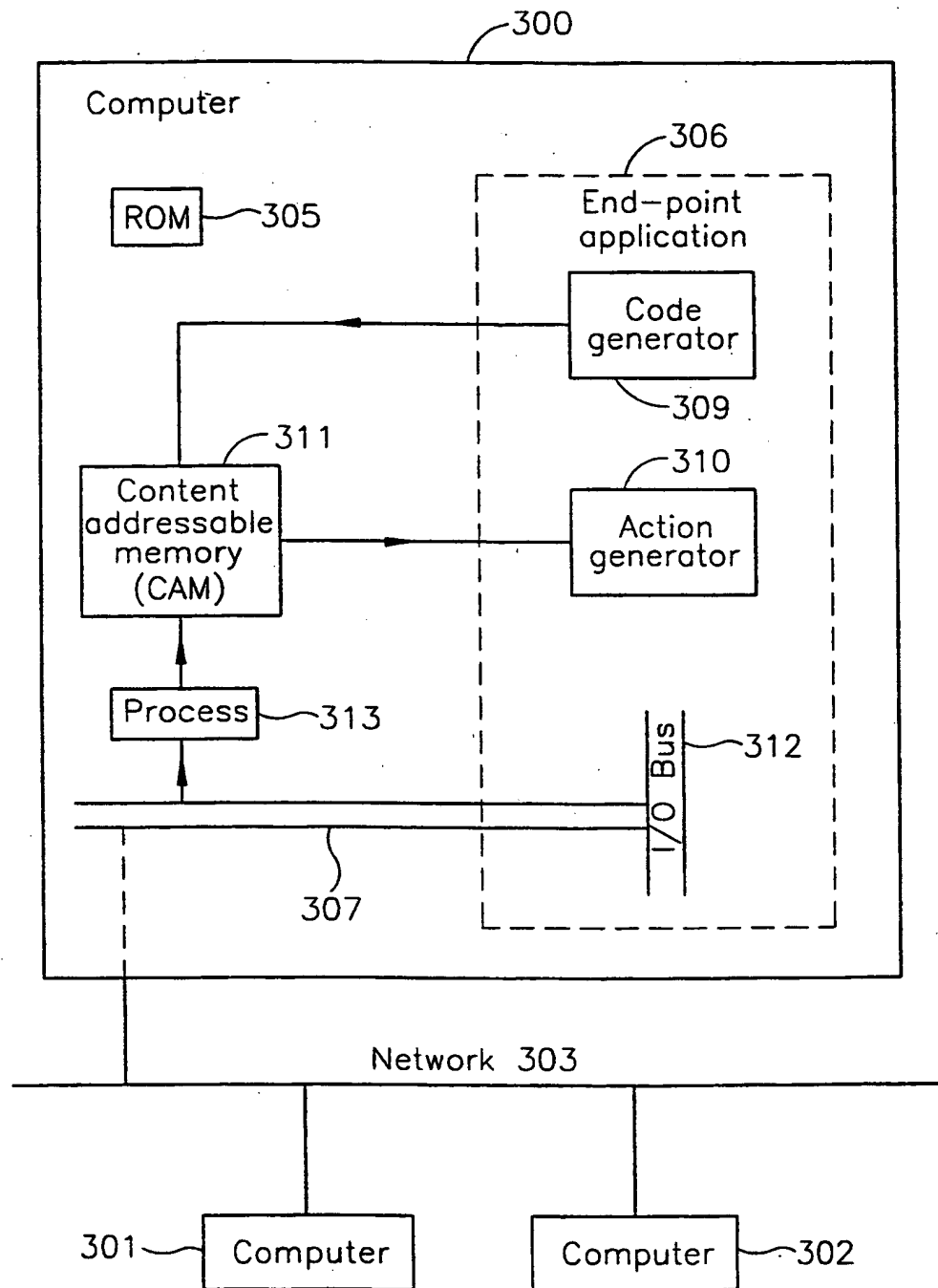


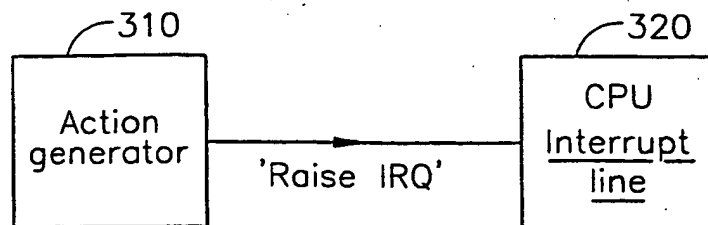
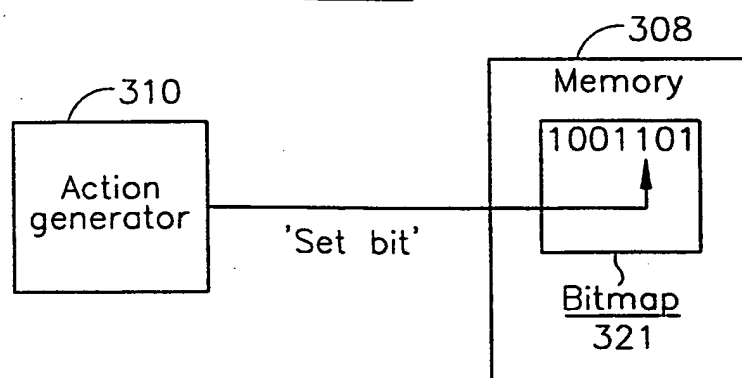
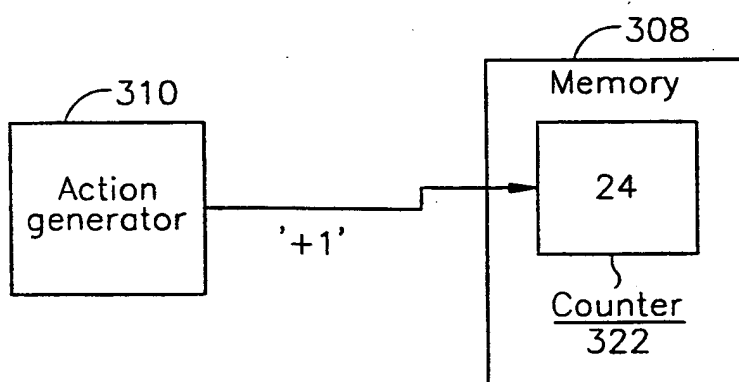
FIG 15

Client-server interaction using a NIC and a hardware data source

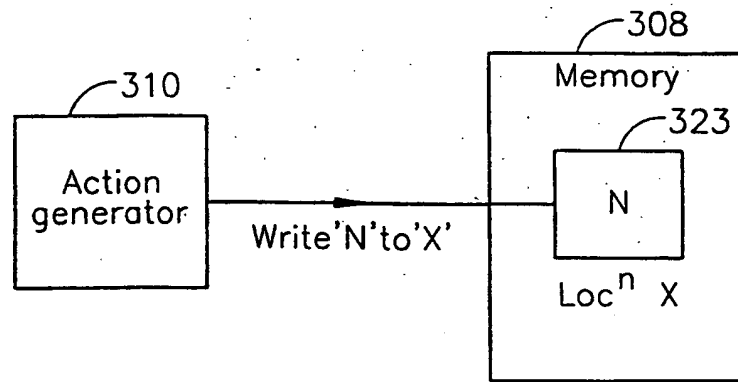
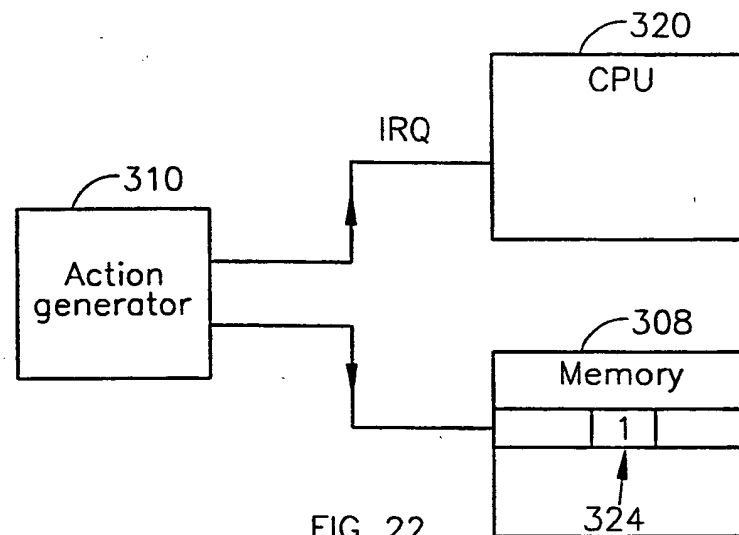
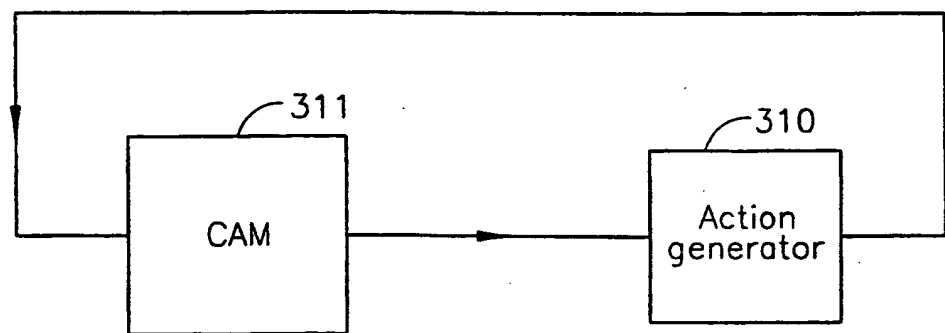
FIG 16

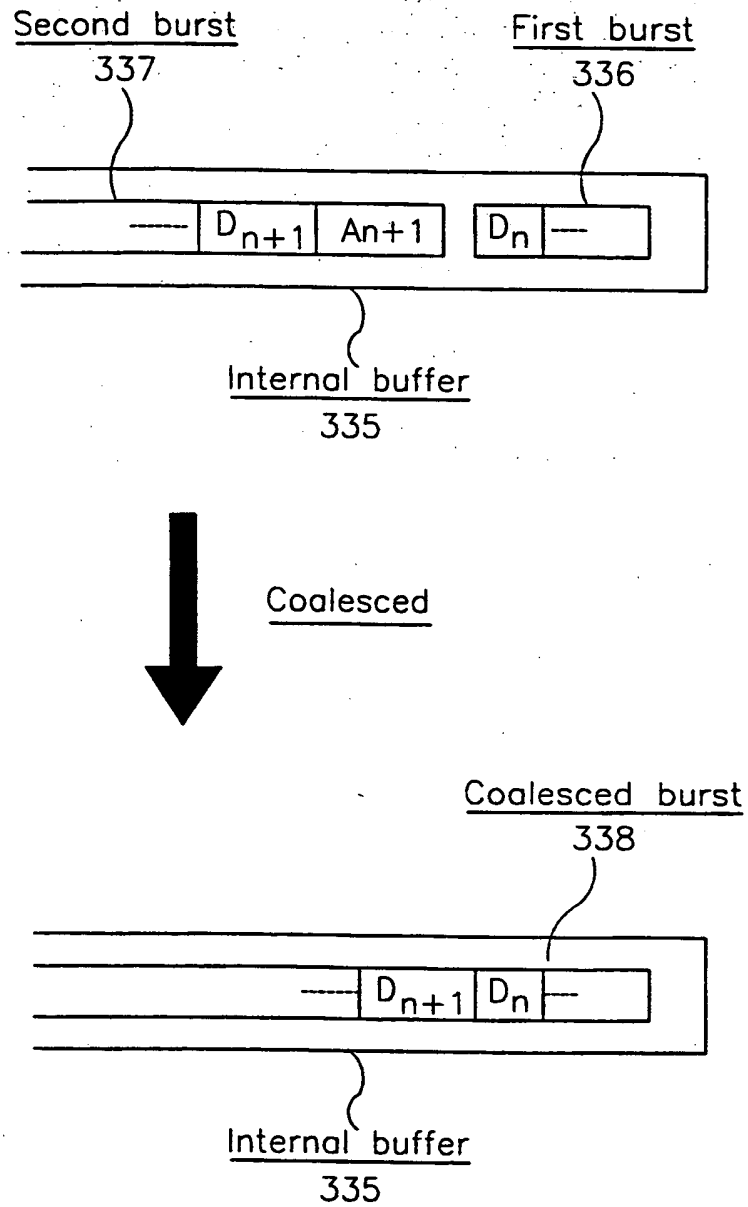
FIG 17

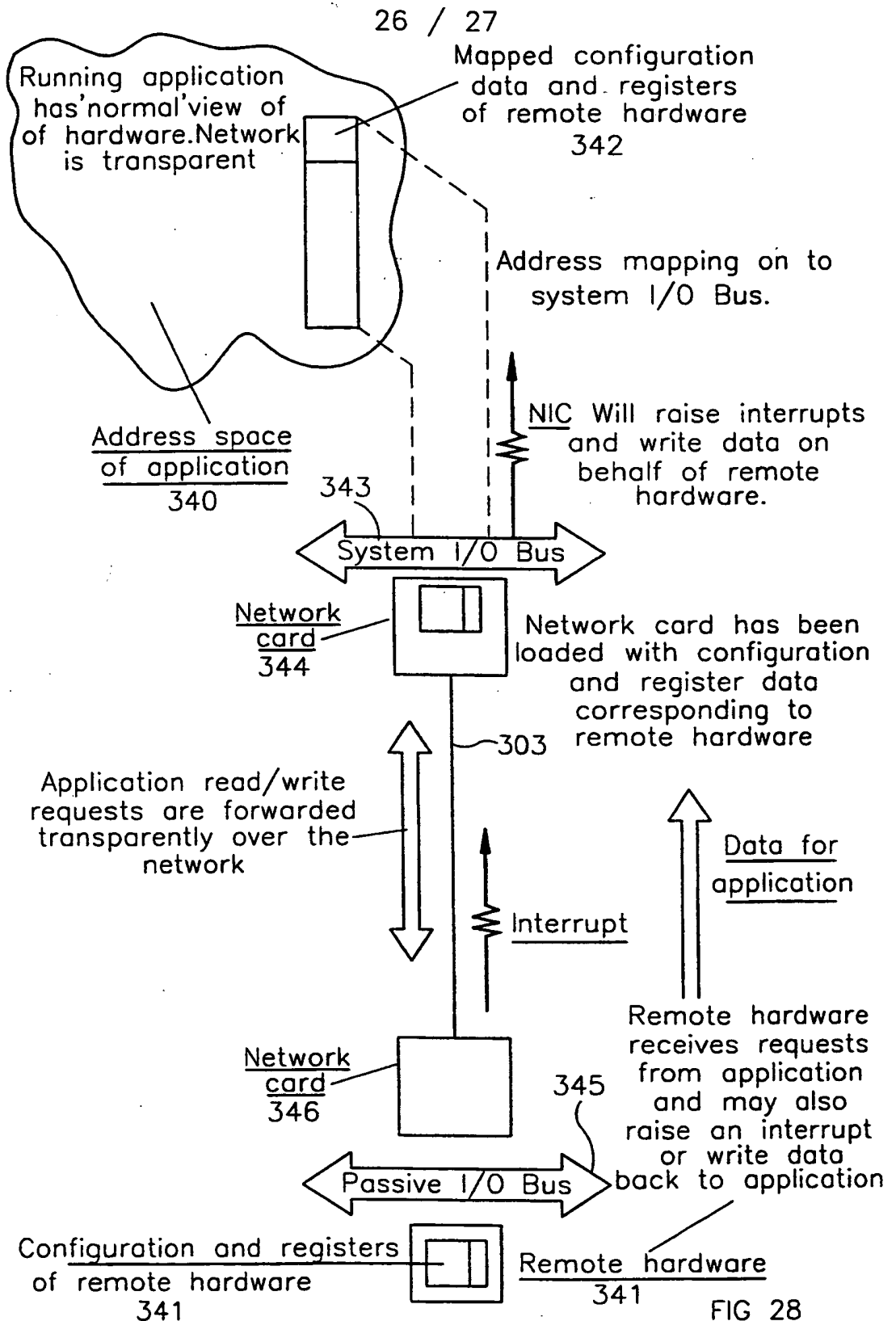
22 / 27

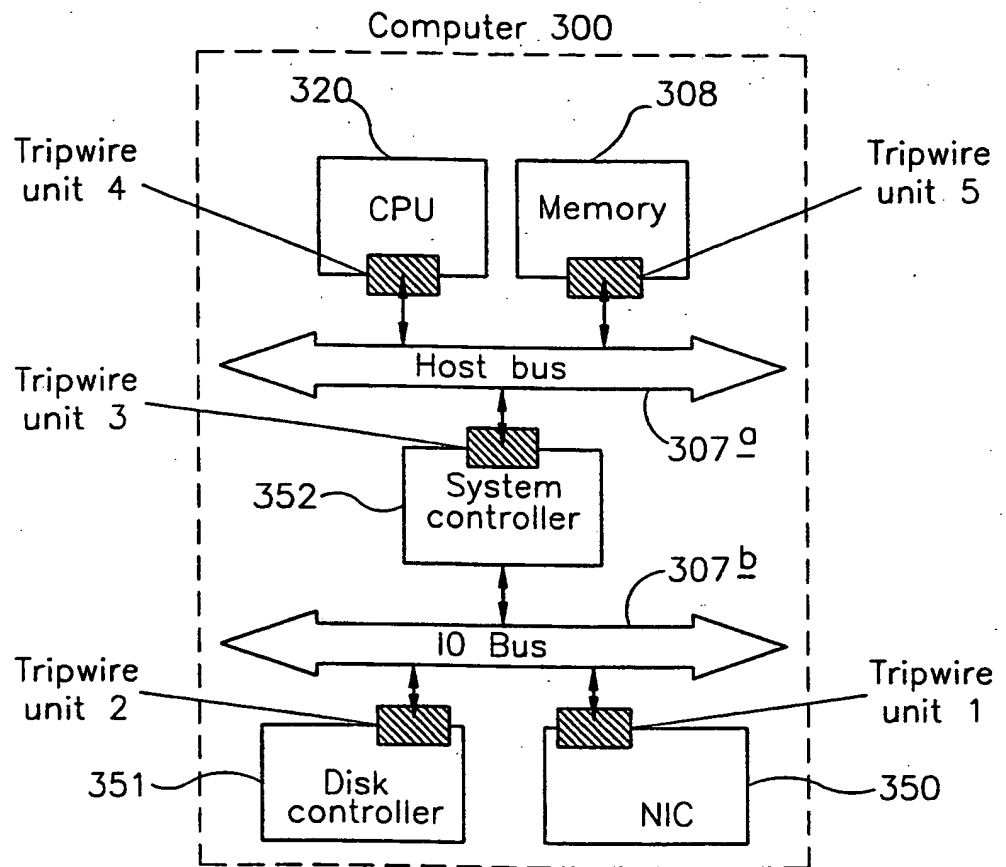
FIG 18FIG 19FIG 20

23 / 27

FIG 21FIG 22FIG 23

FIG 27



FIG 29

European Patent Office
EPA/EPO/OEB
D-80298 Munchen
Germany

JSR.P51122PC

2 August 2001

COPY By Fax/Post

Dear Sirs,

PCT Application No. PCT/GB00/01691
AT&T Laboratories-Cambridge Limited

In response to the Written Opinion dated 3 July 2001, we would offer the following comments on the issues raised by the examiner. The headings of the following sections correspond to those used by the examiner in his detailed report.

Re: Item III

We must disagree with the grouping of various claims suggested by the examiner as relating to a common inventive concept. Independent claims 30 and 45 relate to a specific technique of synchronising applications on two computers connected via a network. The technique has some overlap with the invention claimed in claim 1 of the application but represents a different inventive concept as presented in the claims.

Claim 55 is concerned with transferring data between two ends points where one end point is a computer and the other end point is remote hardware which may be located within a second computer or which may be provided on a passive back plane. Again, there are some similarities between this inventive concept and the inventive concepts of claim 1 and claim 30 but the differences are sufficient to present this concept in a separate set of claims.

Although the independent claims within claims 1 to 71 are concerned with three different inventive concepts such that none of these concepts falls wholly within the scope of either of the other two inventive concepts, nevertheless it is arguable that all of these claims relate to the same basic underlying inventive concept. We would therefore invite the examiner to extend his examination of claims 1 to 29 to claims 30 to 71.

Claims 80 to 113 and 133 to 139 are concerned with a different inventive concept based on a regular queue distributed over a memory mapped network for transferring data between applications. Although the specific inventive concepts, for example in claims 80, 100 and 133, may be used in combination with the inventive concepts of the other independent claims, this is not essential. The individual independent claims of claims 80 to 113 and 133 to 139 concern different inventive concepts but, again, it is at least arguable that these concepts are linked by an underlying single inventive concept.

We have noted the examiner's suggestion to redraft these claims but would respectfully request that this matter be delayed until the national phase of the application. In particular, it is believed that the present format of the claims is in accordance with US practice and it is currently the intention for this application to proceed into the national phase in USA and elsewhere.

Re: Item V

The examiner alleges that various claims are anticipated by the papers by Shaw & Mogul. Shaw relates to inter-process communication and provides several techniques for passing data between processes and/or for synchronising processes. The technique referred to as "semaphore" is generally used for synchronisation and does not normally perform any data transfer. Conversely, the technique referred to as "shared memory" shares data between processes but does not perform any synchronisation by itself. If synchronisation is in addition required, shared memory may be used with semaphore. The techniques referred to as "pipe", "queue" and "device monitor" permit a flow of data only between processes. Synchronisation may also be performed but this is triggered by specific data (including the presence or absence of data) and hence is a "data-based event".

Claim 1 of the present application defines a technique which synchronises an "address-based event" to the presence of a memory address in an information stream comprising data and associated memory addresses. This differs fundamentally from the disclosure of Shaw, in which there is no information stream containing addresses and certainly no memory addresses associated with data.

Mogul discloses a packet filter and relates exclusively to a system in which the packets which are to be delivered are of the "network type" and thus contain an address and data. However, each address is of a "host/port" pair and is not an "associated memory address". The passage at page 40 below Figure 2-2 of Shaw states that "A user process specifies an arbitrary predicate to select the packets it wants". This allows the kernel-resident packet filter to deliver the data to the appropriate application-level buffers. It is implicit that the kernel may reschedule the application so that synchronisation may occur as a result of delivery of a packet to its end point.

However, the information stream in Mogul does not contain "an associated memory address" and so cannot synchronise an action to an address-based event as required by claim 1 of the present application.

The present invention is based on an information stream which is addressed in the same space as "understood" by hardware within a system. This allows the hardware to perform efficient delivery of data to its final destination buffer and synchronisation is based on detection of an address-based event. None of the cited prior art discloses or is capable of this. For example,

some other address space is used (such as "host/port" pair addresses as in Mogul), a mapping must be performed from the addresses in this space to the final buffer locations. This requires additional steps and a typical example of this is the packet filter disclosed in Mogul.

The combination of features defined in claim 1 of the present application is therefore clearly novel with respect to the cited prior art and achieves substantial performance advantages compared with the prior art. We would therefore submit that the cited prior art is not relevant to the patentability of claim 1 and would respectfully request the examiner to re-examine and indicate allowance of claim 1.

The corresponding apparatus claim 132 is therefore, we submit, similarly patentable. Also, because claims 2 to 29 depend on a patentable claim, they are themselves patentable and need not be considered further.

We would prefer to defer commenting on the other claims to which the examiner has objected in this section until the national phase.

Re: Item VII

The examiner suggests that claims 1 to 5 are not properly supported by the description and drawings. However, we would refer the examiner to, for example, Figures 16 and 17 of the drawings and the corresponding description beginning at the third paragraph on page 34 of the specification. The terminology used in claims 1 to 5 is, where possible, also used in the corresponding description so as to clarify the support for these claims. We would therefore submit that the claims are properly supported by the description and strenuously contest the suggestion that the scope of these claims is broader than justified by the description and drawings. In particular, claim 1 defines this inventive concept in terms which are sufficiently broad to give to the applicant the protection to which it is entitled in respect of this invention. We therefore respectfully invite the examiner to withdraw this objection.

As regards the term "end-point application", we appreciate the difficulty of being up-to-date with the terminology used in such a fast-moving technology as that to which the present invention belongs. However, we would confirm that this term is known and has a well-recognised meaning in the art. Further, this term is defined and exemplified in the third paragraph on page 34 of the specification. We therefore believe that the use of this term is permissible in claim 1.

Although we have some sympathy with the examiner's objection to claim 126, nevertheless we believe that this claim may be acceptable in certain jurisdictions and would therefore propose to consider this during the national phase of the application. However, we do not believe that claim 128 is in the form of a definition based on result; the method steps of this claim are clearly defined and create no problem of interpretation of the meaning and scope of this claim, for example to a third party reading the claim and wishing to avoid infringement.

s regards the examiner's final comment on the format of the claims, we propose dealing with this during the national phase as different jurisdictions have different formal requirements for patent claims.

Yours faithfully,

Marks & Clerk

From the:
INTERNATIONAL PRELIMINARY EXAMINING AUTHORITY

To:

ROBINSON, John S.
MARKS & CLERK
4220 Nash Court
Oxford Business Park South
Oxford OX4 2RU
GRANDE BRETAGNE

RECEIVED
- 5 JUL 2001
MARKS AND CLERK

PCT

WRITTEN OPINION

(PCT Rule 66)

Date of mailing (day/month/year)		03.07.2001
Applicant's or agent's file reference JSR.P51122PC		REPLY DUE within 1 month(s) from the above date of mailing
International application No. PCT/GB00/01691	International filing date (day/month/year) 03/05/2000	Priority date (day/month/year) 04/05/1999
International Patent Classification (IPC) or both national classification and IPC G06F13/00		
Applicant AT&T LABORATORIES-CAMBRIDGE LIMITED et al.		

1. This written opinion is the first drawn up by this International Preliminary Examining Authority.
2. This opinion contains indications relating to the following items:
 - I ☒ Basis of the opinion
 - II ☐ Priority
 - III ☒ Non-establishment of opinion with regard to novelty, inventive step and industrial applicability
 - IV ☒ Lack of unity of invention
 - V ☒ Reasoned statement under Rule 66.2(a)(ii) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement
 - VI ☐ Certain document cited
 - VII ☐ Certain defects in the international application
 - VIII ☒ Certain observations on the international application
3. The applicant is hereby invited to reply to this opinion.


When? See the time limit indicated above. The applicant may, before the expiration of that time limit, request this Authority to grant an extension, see Rule 66.2(d).

How? By submitting a written reply, accompanied, where appropriate, by amendments, according to Rule 66.3. For the form and the language of the amendments, see Rules 66.8 and 66.9.

Also: For an additional opportunity to submit amendments, see Rule 66.4.
For the examiner's obligation to consider amendments and/or arguments, see Rule 66.4 bis.
For an informal communication with the examiner, see Rule 66.6.

If no reply is filed, the international preliminary examination report will be established on the basis of this opinion.
4. The final date by which the international preliminary examination report must be established according to Rule 69.2 is: 04/09/2001.

Name and mailing address of the international preliminary examining authority:

 European Patent Office
D-80298 Munich
Tel. +49 89 2399 - 0 Tx: 523656 epmu d
Fax: +49 89 2399 - 4465

Authorized officer / Examiner

Anastassiades, G

Formalities officer (incl. extension of time limits)

Muehlbauer, P

Telephone No. +49 89 2399 2513



I. Basis of the opinion

1. With regard to the elements of the international application (*Replacement sheets which have been furnished to the receiving Office in response to an invitation under Article 14 are referred to in this opinion as "originally filed"*):

Description, pages:

1-42 as originally filed

Claims, No.:

1-139 as originally filed

Drawings, sheets:

1/27-27/27 as originally filed

2. With regard to the language, all the elements marked above were available or furnished to this Authority in the language in which the international application was filed, unless otherwise indicated under this item.

These elements were available or furnished to this Authority in the following language: , which is:

- ☐ the language of a translation furnished for the purposes of the international search (under Rule 23.1(b)).
- ☐ the language of publication of the international application (under Rule 48.3(b)).
- ☐ the language of a translation furnished for the purposes of international preliminary examination (under Rule 55.2 and/or 55.3).

3. With regard to any nucleotide and/or amino acid sequence disclosed in the international application, the international preliminary examination was carried out on the basis of the sequence listing:

- ☐ contained in the international application in written form.
- ☐ filed together with the international application in computer readable form.
- ☐ furnished subsequently to this Authority in written form.
- ☐ furnished subsequently to this Authority in computer readable form.
- ☐ The statement that the subsequently furnished written sequence listing does not go beyond the disclosure in the international application as filed has been furnished.
- ☐ The statement that the information recorded in computer readable form is identical to the written sequence listing has been furnished.

4. The amendments have resulted in the cancellation of:

- ☐ the description, pages:
- ☐ the claims, Nos.:

☐ the drawings, sheets:

5. ☐ This report has been established as if (some of) the amendments had not been made, since they have been considered to go beyond the disclosure as filed (Rule 70.2(c)):

(Any replacement sheet containing such amendments must be referred to under item 1 and annexed to this report.)

6. Additional observations, if necessary:

III. Non-establishment of opinion with regard to novelty, inventive step and industrial applicability

1. The questions whether the claimed invention appears to be novel, to involve an inventive step (to be non-obvious), or to be industrially applicable have not been and will not be examined in respect of:

☐ the entire international application,

☒ claims Nos. 30, 55, 80, 133,

because:

☐ the said international application, or the said claims Nos. relate to the following subject matter which does not require an international preliminary examination (*specify*):

☐ the description, claims or drawings (*indicate particular elements below*) or said claims Nos. are so unclear that no meaningful opinion could be formed (*specify*):

☐ the claims, or said claims Nos. are so inadequately supported by the description that no meaningful opinion could be formed.

☐ no international search report has been established for the said claims Nos. .

2. A written opinion cannot be drawn due to the failure of the nucleotide and/or amino acid sequence listing to comply with the standard provided for in Annex C of the Administrative Instructions:

☐ the written form has not been furnished or does not comply with the standard.

☐ the computer readable form has not been furnished or does not comply with the standard.

IV. Lack of unity of invention

1. In response to the invitation (Form PCT/IPEA/405) to restrict or pay additional fees, the applicant has:

☐ restricted the claims.

☒ paid additional fees.

- ☐ paid additional fees under protest.
- ☐ neither restricted nor paid additional fees.
- 2. ☐ This Authority found that the requirement of unity of invention is not complied with for the following reasons and chose, according to Rule 68.1, not to invite the applicant to restrict or pay additional fees:
- 3. Consequently, the following parts of the international application were the subject of international preliminary examination in establishing this opinion:
 - ☒ all parts.
 - ☐ the parts relating to claims Nos. .

V. Reasoned statement under Rule 66.2(a)(ii) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

1. Statement
- | | | |
|-------------------------------|--------|--------------------------------|
| Novelty (N) | Claims | 1-5, 27-29, 126, 128, 129, 132 |
| Inventive step (IS) | Claims | |
| Industrial applicability (IA) | Claims | |

2. Citations and explanations
see separate sheet

VIII. Certain observations on the international application

The following observations on the clarity of the claims, description, and drawings or on the question whether the claims are fully supported by the description, are made:
see separate sheet

Re Item III

Non-establishment of opinion with regard to novelty, inventive step and industrial applicability

Concerning claims 30-44, 45-54, 55-71, 80-99, 100-113 and 133-139

1. Although claims 30, 55, 80 and 133 have been drafted as separate independent claims, they appear to relate effectively to the same subject-matter and to differ from each other only with regard to the definition of the subject-matter for which protection is sought or in respect of the terminology used for the features of that subject-matter. The aforementioned claims therefore lack conciseness. Moreover, lack of clarity of the claims as a whole arises, since the plurality of independent claims makes it difficult, if not impossible, to identify the claimed invention, and places an undue burden on others seeking to establish the extent of the protection.

Hence, claims 30, 55, 80 and 133 do not meet the requirements of Article 6 PCT.

2. In order to overcome this objection, it would appear appropriate to file an amended set of claims defining the relevant subject-matter in terms of a minimum number of independent method claims (for example, two method claims corresponding to apparatus claims 45 and 100 respectively) followed by dependent claims covering features which are merely optional (Rule 6.4 PCT).

Re Item V

Reasoned statement under Rule 66.2(a)(ii) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

Concerning claims 1-29 and 132

1. The broad wording of claim 1 includes synchronising methods such as known from document SHAW R H: "A COMPLETE GUIDE TO OS/2 INTERPROCESS COMMUNICATIONS AND DEVICE MONITORS" MICROSOFT SYSTEMS

JOURNAL,US,MICROSOFT CO., REDMOND, WA, vol. 4, no. 5, 1 September 1989 (1989-09-01), pages 35-60, XP000568106 ISSN: 0889-9932 or MOGUL J C ET AL: "THE PACKET FILTER: AN EFFICIENT MECHANISM FOR USER-LEVEL NETWORK CODE" OPERATING SYSTEMS REVIEW (SIGOPS),US,ACM HEADQUARTER. NEW YORK, 1987, pages 39-51, XP002913603.

2. The additional features of claims 2-5 are also known from the R H Shaw document mentioned in the previous paragraph.
3. The combination of the features of dependent claim 6 is neither known from, nor rendered obvious by, the available prior art. It is suggested therefore that a new independent claim be drafted to include these features.
4. Independent claims 27-29 refer to claim 1. Independent apparatus claim 132 corresponds to method claim 1. Therefore, all observations in paragraphs 1-3 above apply correspondingly to these independent claims.

Concerning claims 114-125

1. All features of claim 1 are known from document EP 0 784 268 A (see Abstract; section "System Architecture" on pages 4 and 5). The claim therefore lacks novelty.
2. The additional features of claim 115 refer to unexamined claims 80-99 (see Item III above), which relate to another invention.
3. The additional features of dependent claims 116-125 are either known from, or rendered obvious by, the prior art documents EP 0 784 268 A and US 5 787 251 A.

Concerning claims 126, 127, 128 and 129-131

The features of claims 126 and 128 that are adequately defined (see Item IIIV below) and of claim 129 appear to be known from document EP 0 772 368 A.

Re Item VII

Certain defects in the international application

Re Item VIII

Certain observations on the international application

Concerning claims 1-29 and 132

1. Claims 1-5 are not supported by the description as required by Article 6 PCT, as their scope is broader than justified by the description and drawings.
2. The term "end-point application" used in claim 1 does not appear to have a well-recognised meaning in the art. The term should therefore be clarified.

Concerning claims 126, 127, 128 and 129-131

Claims 126 and 128 do not meet the requirements of Article 6 PCT in that the matter for which protection is sought is not clearly defined. The claims attempt to define the subject-matter in terms of the result to be achieved (the destination addresses being inferable in claim 126 and calculation of the addresses of each data word in claim 128), which merely amounts to a statement of the underlying problem. The technical features necessary for achieving this result should be added (see, for example, description page 39, second paragraph)..

Concerning all claims

1. The Independent claims are not in the two-part form in accordance with Rule 6.3(b) PCT, which in the present case would be appropriate, with those features known in combination from the prior art being placed in the preamble (Rule 6.3(b)(i) PCT) and with the remaining features being included in the characterising part (Rule 6.3(b)(ii) PCT).
2. The features of the claims are not provided with reference signs placed in parentheses (Rule 6.2(b) PCT).

From the
INTERNATIONAL PRELIMINARY EXAMINING AUTHORITY

To:

ROBINSON, John S.
MARKS & CLERK
4220 Nash Court
Oxford Business Park South
Oxford OX4 2RU
GRANDE BRETAGNE

PCT

NOTIFICATION OF TRANSMITTAL OF
THE INTERNATIONAL PRELIMINARY
EXAMINATION REPORT

(PCT Rule 71.1)

Date of mailing
(day/month/year)

31.08.2001

Applicant's or agent's file reference
JSR.P51122PC

IMPORTANT NOTIFICATION

International application No.
PCT/GB00/01691

International filing date (day/month/year)
03/05/2000

Priority date (day/month/year)
04/05/1999

Applicant

AT&T LABORATORIES-CAMBRIDGE LIMITED et al.

1. The applicant is hereby notified that this International Preliminary Examining Authority transmits herewith the international preliminary examination report and its annexes, if any, established on the international application.
2. A copy of the report and its annexes, if any, is being transmitted to the International Bureau for communication to all the elected Offices.
3. Where required by any of the elected Offices, the International Bureau will prepare an English translation of the report (but not of any annexes) and will transmit such translation to those Offices.

4. REMINDER

The applicant must enter the national phase before each elected Office by performing certain acts (filing translations and paying national fees) within 30 months from the priority date (or later in some Offices) (Article 39(1)) (see also the reminder sent by the International Bureau with Form PCT/IB/301).

Where a translation of the international application must be furnished to an elected Office, that translation must contain a translation of any annexes to the international preliminary examination report. It is the applicant's responsibility to prepare and furnish such translation directly to each elected Office concerned.

For further details on the applicable time limits and requirements of the elected Offices, see Volume II of the PCT Applicant's Guide.

Name and mailing address of the IPEA/



European Patent Office
D-80298 Munich
Tel. +49 89 2399 - 0 Tx: 523658 epmu d
Fax: +49 89 2399 - 4465

Authorized officer

Koski, P



Tel. +49 89 2399-2709



PCT

INTERNATIONAL PRELIMINARY EXAMINATION REPORT

(PCT Article 36 and Rule 70)

Applicant's or agent's file reference JSR.P51122PC		FOR FURTHER ACTION See Notification of Transmittal of International Preliminary Examination Report (Form PCT/IPEA/416)	
International application No. PCT/GB00/01691	International filing date (day/month/year) 03/05/2000	Priority date (day/month/year) 04/05/1999	
International Patent Classification (IPC) or national classification and IPC G06F13/00			
Applicant AT&T LABORATORIES-CAMBRIDGE LIMITED et al.			
<p>1. This international preliminary examination report has been prepared by this International Preliminary Examining Authority and is transmitted to the applicant according to Article 36.</p> <p>2. This REPORT consists of a total of 9 sheets, including this cover sheet.</p> <p><input type="checkbox"/> This report is also accompanied by ANNEXES, i.e. sheets of the description, claims and/or drawings which have been amended and are the basis for this report and/or sheets containing rectifications made before this Authority (see Rule 70.16 and Section 607 of the Administrative Instructions under the PCT).</p> <p>These annexes consist of a total of sheets.</p>			
<p>3. This report contains indications relating to the following items:</p> <ul style="list-style-type: none"> I <input checked="" type="checkbox"/> Basis of the report II <input type="checkbox"/> Priority III <input checked="" type="checkbox"/> Non-establishment of opinion with regard to novelty, inventive step and industrial applicability IV <input checked="" type="checkbox"/> Lack of unity of invention V <input checked="" type="checkbox"/> Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement VI <input type="checkbox"/> Certain documents cited VII <input type="checkbox"/> Certain defects in the international application VIII <input checked="" type="checkbox"/> Certain observations on the international application 			
Date of submission of the demand 15/11/2000		Date of completion of this report 31.08.2001	
Name and mailing address of the international preliminary examining authority:  European Patent Office D-80298 Munich Tel. +49 89 2399 - 0 Tx: 523658 epmu d Fax: +49 89 2399 - 4485		Authorized officer Anastassiades, G Telephone No. +49 89 2399 2497 	

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT**

International application No. PCT/GB00/01691

I. Basis of the report

1. With regard to the elements of the international application (*Replacement sheets which have been furnished to the receiving Office in response to an invitation under Article 14 are referred to in this report as "originally filed" and are not annexed to this report since they do not contain amendments (Rules 70.16 and 70.17):*)

Description, pages:

1-42 as originally filed

Claims, No.:

1-139 as originally filed

Drawings, sheets:

1/27-27/27 as originally filed

2. With regard to the language, all the elements marked above were available or furnished to this Authority in the language in which the international application was filed, unless otherwise indicated under this item.

These elements were available or furnished to this Authority in the following language: , which is:

- ☐ the language of a translation furnished for the purposes of the international search (under Rule 23.1(b)).
- ☐ the language of publication of the international application (under Rule 48.3(b)).
- ☐ the language of a translation furnished for the purposes of international preliminary examination (under Rule 55.2 and/or 55.3).

3. With regard to any nucleotide and/or amino acid sequence disclosed in the international application, the international preliminary examination was carried out on the basis of the sequence listing:

- ☐ contained in the international application in written form.
- ☐ filed together with the international application in computer readable form.
- ☐ furnished subsequently to this Authority in written form.
- ☐ furnished subsequently to this Authority in computer readable form.
- ☐ The statement that the subsequently furnished written sequence listing does not go beyond the disclosure in the international application as filed has been furnished.
- ☐ The statement that the information recorded in computer readable form is identical to the written sequence listing has been furnished.

4. The amendments have resulted in the cancellation of:

- ☐ the description, pages:
- ☐ the claims, Nos.:

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT**

International application No. PCT/GB00/01691

- ☐ the drawings, sheets;
5. ☐ This report has been established as if (some of) the amendments had not been made, since they have been considered to go beyond the disclosure as filed (Rule 70.2(c)):
(Any replacement sheet containing such amendments must be referred to under item 1 and annexed to this report.)
6. Additional observations, if necessary:

III. Non-establishment of opinion with regard to novelty, inventive step and industrial applicability

1. The questions whether the claimed invention appears to be novel, to involve an inventive step (to be non-obvious), or to be industrially applicable have not been examined in respect of:

- ☐ the entire international application.
- ☒ claims Nos. 30-44, 45-54, 55-71, 80-99, 100-113, 133-139.

because:

- ☐ the said international application, or the said claims Nos. relate to the following subject matter which does not require an international preliminary examination (*specify*):
- ☒ the description, claims or drawings (*indicate particular elements below*) or said claims Nos. are so unclear that no meaningful opinion could be formed (*specify*):
see separate sheet
- ☐ the claims, or said claims Nos. are so inadequately supported by the description that no meaningful opinion could be formed.
- ☐ no international search report has been established for the said claims Nos. .
2. A meaningful international preliminary examination cannot be carried out due to the failure of the nucleotide and/or amino acid sequence listing to comply with the standard provided for in Annex C of the Administrative Instructions:
- ☐ the written form has not been furnished or does not comply with the standard.
- ☐ the computer readable form has not been furnished or does not comply with the standard.

IV. Lack of unity of invention

1. In response to the invitation to restrict or pay additional fees the applicant has:

- ☐ restricted the claims.

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT**

International application No. PCT/GB00/01691

- ☒ paid additional fees.
- ☐ paid additional fees under protest.
- ☐ neither restricted nor paid additional fees.
2. ☐ This Authority found that the requirement of unity of invention is not complied and chose, according to Rule 68.1, not to invite the applicant to restrict or pay additional fees.
3. This Authority considers that the requirement of unity of invention in accordance with Rules 13.1, 13.2 and 13.3 is
- ☐ complied with.
- ☐ not complied with for the following reasons:
4. Consequently, the following parts of the international application were the subject of international preliminary examination in establishing this report:
- ☒ all parts.
- ☐ the parts relating to claims Nos. .

V. Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

1. Statement

Novelty (N)	Yes: Claims 6-26, 72-79
	No: Claims 1-5, 27-29, 132, 114, 115, 126, 128, 129
Inventive step (IS)	Yes: Claims 6-26, 72-79
	No: Claims 116-125
Industrial applicability (IA)	Yes: Claims 1-139
	No: Claims

**2. Citations and explanations
see separate sheet**

VIII. Certain observations on the international application

The following observations on the clarity of the claims, description, and drawings or on the question whether the claims are fully supported by the description, are made:
see separate sheet

Re Item III

Non-establishment of opinion with regard to novelty, inventive step and industrial applicability

Concerning claims 30-44, 45-54, 55-71, 80-99, 100-113 and 133-139

1. Although claims 30, 55, 80 and 133 have been drafted as separate independent claims, they appear to relate effectively to the same subject-matter and to differ from each other only with regard to the definition of the subject-matter for which protection is sought or in respect of the terminology used for the features of that subject-matter. The aforementioned claims therefore lack conciseness. Moreover, lack of clarity of the claims as a whole arises, since the plurality of independent claims makes it difficult, if not impossible, to identify the claimed invention, and places an undue burden on others seeking to establish the extent of the protection.

Hence, claims 30, 55, 80 and 133 do not meet the requirements of Article 6 PCT.

Re Item IV

Lack of unity of invention

1. Claims 1-29 and 132 relate to method and apparatus for synchronising an end-point application in a computer, involving the detection of an address-based event in an Information stream.
2. Claims 30-44, 45-54, 55-71, 80-99, 100-113 and 133-139 relate to method and apparatus for directing incoming data exchanged between two computers involving establishing control values for allowable locations for placing incoming data.
3. Claims 72-79 relate to a method of arranging data transfers on a computer involving a DMA engine.

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

4. Claims 114-125 relate to a method of sending a request from a client application on a first computer to a server application on a second computer and a response thereto in an opposite direction, involving providing software stubs which produce a marshalled stream of data.
5. Claims 126, 127, 128 and 129-131 relate to a method of arranging data for transfer as a data burst over a computer network involving providing a particular header address, allowing halting transfer of a data burst.

There is no technical relationship among the inventions listed above involving one or more of the same or corresponding special technical features in the sense of Rule 13.2 PCT.

Re Item V

Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

Concerning claims 1-29 and 132

1. The broad wording of claim 1 includes synchronising methods such as known from document SHAW R H: "A COMPLETE GUIDE TO OS/2 INTERPROCESS COMMUNICATIONS AND DEVICE MONITORS" MICROSOFT SYSTEMS JOURNAL, US, MICROSOFT CO., REDMOND, WA, vol. 4, no. 5, 1 September 1989 (1989-09-01), pages 35-60, XP000568106 ISSN: 0889-9932 or MOGUL J C ET AL: "THE PACKET FILTER: AN EFFICIENT MECHANISM FOR USER-LEVEL NETWORK CODE" OPERATING SYSTEMS REVIEW (SIGOPS), US, ACM HEADQUARTER. NEW YORK, 1987, pages 39-51, XP002913603.
2. The additional features of claims 2-5 are also known from the R H Shaw document mentioned in the previous paragraph.
3. The combination of the features of dependent claim 6 is neither known from, nor

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

rendered obvious by, the available prior art. Claim 6 therefore meets the requirements of Article 33(2) and (3) PCT.

4. Dependent claims 7-26, so far as including claim 6, also meet the requirements of Article 33(2) and (3) PCT.
4. Independent claims 27-29 refer to claim 1. Independent apparatus claim 132 corresponds to method claim 1. Therefore, all observations in paragraphs 1-3 above apply correspondingly to these independent claims.

Concerning claims 72-79

1. The use of triggering values for arranging DMA data transfers as set out in independent method claim 72 is not obvious nor known from the prior art. Claim 72 therefore meets the requirements of Article 33(2) and (3) PCT.
2. Claims 72-79 are dependent from claim 2 and therefore also meet the requirement of Article 33(2) and (3) PCT.

Concerning claims 114-125

1. All features of claim 114 are known from document EP 0 784 268 A (see Abstract; section "System Architecture" on pages 4 and 5). The claim therefore lacks novelty.
2. The additional features of claim 115 refer to unexamined claims 80-99 (see Item III above), which relate to another invention.
3. The additional features of dependent claims 116-125 are either known from, or rendered obvious by, the prior art documents EP 0 784 268 A and US 5 787 251 A.

Concerning claims 126, 127, 128 and 129-131

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

The features of claims 126 and 128 that are adequately defined (see Item IIIV below) and of claim 129 appear to be known from document EP 0 772 368 A.

Re Item VIII

Certain observations on the international application

Concerning claims 1-29 and 132

1. Claims 1-5 are not supported by the description as required by Article 6 PCT, as their scope is broader than justified by the description and drawings.
2. The term "end-point application" used in claim 1 does not appear to have a well-recognised meaning in the art. The term should therefore have been clarified.

Concerning claims 126, 127, 128 and 129-131

Claims 126 and 128 do not meet the requirements of Article 6 PCT in that the matter for which protection is sought is not clearly defined. The claims attempt to define the subject-matter in terms of the result to be achieved (the destination addresses being inferable in claim 126 and calculation of the addresses of each data word in claim 128), which merely amounts to a statement of the underlying problem. The technical features necessary for achieving this result should have been added (see, for example, description page 39, second paragraph)..

Concerning all claims

1. The Independent claims are not in the two-part form in accordance with Rule 6.3(b) PCT, which in the present case would be appropriate, with those features known in combination from the prior art being placed in the preamble (Rule 6.3(b)(i) PCT) and with the remaining features being included in the characterising part (Rule 6.3(b)(ii) PCT).

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

2. The features of the claims are not provided with reference signs placed in parentheses (Rule 6.2(b) PCT).

PATENT COOPERATION TREATY

PCT


REC'D 04 SEP 2001

WIPO

PCT

INTERNATIONAL PRELIMINARY EXAMINATION REPORT

(PCT Article 36 and Rule 70)

Applicant's or agent's file reference JSR.P51122PC		FOR FURTHER ACTION	See Notification of Transmittal of International Preliminary Examination Report (Form PCT/IPEA/416)
International application No. PCT/GB00/01691	International filing date (day/month/year) 03/05/2000	Priority date (day/month/year) 04/05/1999	
International Patent Classification (IPC) or national classification and IPC G06F13/00			
Applicant AT&T LABORATORIES-CAMBRIDGE LIMITED et al.			
<p>1. This international preliminary examination report has been prepared by this International Preliminary Examining Authority and is transmitted to the applicant according to Article 36.</p> <p>2. This REPORT consists of a total of 9 sheets, including this cover sheet.</p> <p><input type="checkbox"/> This report is also accompanied by ANNEXES, i.e. sheets of the description, claims and/or drawings which have been amended and are the basis for this report and/or sheets containing rectifications made before this Authority (see Rule 70.16 and Section 607 of the Administrative Instructions under the PCT).</p> <p>These annexes consist of a total of sheets.</p>			
<p>3. This report contains indications relating to the following items:</p> <ul style="list-style-type: none"> I <input checked="" type="checkbox"/> Basis of the report II <input type="checkbox"/> Priority III <input checked="" type="checkbox"/> Non-establishment of opinion with regard to novelty, inventive step and industrial applicability IV <input checked="" type="checkbox"/> Lack of unity of invention V <input checked="" type="checkbox"/> Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement VI <input type="checkbox"/> Certain documents cited VII <input type="checkbox"/> Certain defects in the international application VIII <input checked="" type="checkbox"/> Certain observations on the international application 			
Date of submission of the demand 15/11/2000		Date of completion of this report 31.08.2001	
Name and mailing address of the international preliminary examining authority:  European Patent Office D-80298 Munich Tel. +49 89 2399 - 0 Tx: 523656 epmu d Fax: +49 89 2399 - 4465		Authorized officer Anastassiades, G Telephone No. +49 89 2399 2497	



**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT**

International application No. PCT/GB00/01691

I. Basis of the report

1. With regard to the elements of the international application (*Replacement sheets which have been furnished to the receiving Office in response to an invitation under Article 14 are referred to in this report as "originally filed" and are not annexed to this report since they do not contain amendments (Rules 70.16 and 70.17)*):

Description, pages:

1-42 as originally filed

Claims, No.:

1-139 as originally filed

Drawings, sheets:

1/27-27/27 as originally filed

2. With regard to the **language**, all the elements marked above were available or furnished to this Authority in the language in which the international application was filed, unless otherwise indicated under this item.

These elements were available or furnished to this Authority in the following language: , which is:

- ☐ the language of a translation furnished for the purposes of the international search (under Rule 23.1(b)).
- ☐ the language of publication of the international application (under Rule 48.3(b)).
- ☐ the language of a translation furnished for the purposes of international preliminary examination (under Rule 55.2 and/or 55.3).

3. With regard to any **nucleotide and/or amino acid sequence** disclosed in the international application, the international preliminary examination was carried out on the basis of the sequence listing:

- ☐ contained in the international application in written form.
- ☐ filed together with the international application in computer readable form.
- ☐ furnished subsequently to this Authority in written form.
- ☐ furnished subsequently to this Authority in computer readable form.
- ☐ The statement that the subsequently furnished written sequence listing does not go beyond the disclosure in the international application as filed has been furnished.
- ☐ The statement that the information recorded in computer readable form is identical to the written sequence listing has been furnished.

4. The amendments have resulted in the cancellation of:

- ☐ the description, pages:
- ☐ the claims, Nos.:

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT**

International application No. PCT/GB00/01691

☐ the drawings, sheets:

5. ☐ This report has been established as if (some of) the amendments had not been made, since they have been considered to go beyond the disclosure as filed (Rule 70.2(c)):

(Any replacement sheet containing such amendments must be referred to under item 1 and annexed to this report.)

6. Additional observations, if necessary:

III. Non-establishment of opinion with regard to novelty, inventive step and industrial applicability

1. The questions whether the claimed invention appears to be novel, to involve an inventive step (to be non-obvious), or to be industrially applicable have not been examined in respect of:

☐ the entire international application.

☒ claims Nos. 30-44,45-54,55-71,80-99,100-113,133-139.

because:

☐ the said international application, or the said claims Nos. relate to the following subject matter which does not require an international preliminary examination (*specify*):

☒ the description, claims or drawings (*indicate particular elements below*) or said claims Nos. are so unclear that no meaningful opinion could be formed (*specify*):
see separate sheet

☐ the claims, or said claims Nos. are so inadequately supported by the description that no meaningful opinion could be formed.

☐ no international search report has been established for the said claims Nos. .

2. A meaningful international preliminary examination cannot be carried out due to the failure of the nucleotide and/or amino acid sequence listing to comply with the standard provided for in Annex C of the Administrative Instructions:

☐ the written form has not been furnished or does not comply with the standard.

☐ the computer readable form has not been furnished or does not comply with the standard.

IV. Lack of unity of invention

1. In response to the invitation to restrict or pay additional fees the applicant has:

☐ restricted the claims.

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT**

International application No. PCT/GB00/01691

- ☒ paid additional fees.
- ☐ paid additional fees under protest.
- ☐ neither restricted nor paid additional fees.

2. ☐ This Authority found that the requirement of unity of invention is not complied and chose, according to Rule 68.1, not to invite the applicant to restrict or pay additional fees.
3. This Authority considers that the requirement of unity of invention in accordance with Rules 13.1, 13.2 and 13.3 is
- ☐ complied with.
 - ☐ not complied with for the following reasons:
4. Consequently, the following parts of the international application were the subject of international preliminary examination in establishing this report:
- ☒ all parts.
 - ☐ the parts relating to claims Nos. .

V. Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

1. Statement

Novelty (N)	Yes:	Claims 6-26, 72-79
	No:	Claims 1-5, 27-29, 132, 114, 115, 126, 128, 129
Inventive step (IS)	Yes:	Claims 6-26, 72-79
	No:	Claims 116-125
Industrial applicability (IA)	Yes:	Claims 1-139
	No:	Claims

**2. Citations and explanations
see separate sheet**

VIII. Certain observations on the international application

The following observations on the clarity of the claims, description, and drawings or on the question whether the claims are fully supported by the description, are made:
see separate sheet

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

Re Item III

Non-establishment of opinion with regard to novelty, inventive step and industrial applicability

Concerning claims 30-44, 45-54, 55-71, 80-99, 100-113 and 133-139

1. Although claims 30, 55, 80 and 133 have been drafted as separate independent claims, they appear to relate effectively to the same subject-matter and to differ from each other only with regard to the definition of the subject-matter for which protection is sought or in respect of the terminology used for the features of that subject-matter. The aforementioned claims therefore lack conciseness. Moreover, lack of clarity of the claims as a whole arises, since the plurality of independent claims makes it difficult, if not impossible, to identify the claimed invention, and places an undue burden on others seeking to establish the extent of the protection.

Hence, claims 30, 55, 80 and 133 do not meet the requirements of Article 6 PCT.

Re Item IV

Lack of unity of invention

1. Claims 1-29 and 132 relate to method and apparatus for synchronising an end-point application in a computer, involving the detection of an address-based event in an information stream.
2. Claims 30-44, 45-54, 55-71, 80-99, 100-113 and 133-139 relate to method and apparatus for directing incoming data exchanged between two computers involving establishing control values for allowable locations for placing incoming data.
3. Claims 72-79 relate to a method of arranging data transfers on a computer involving a DMA engine.

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

4. Claims 114-125 relate to a method of sending a request from a client application on a first computer to a server application on a second computer and a response thereto in an opposite direction, involving providing software stubs which produce a marshalled stream of data.
5. Claims 126, 127, 128 and 129-131 relate to a method of arranging data for transfer as a data burst over a computer network involving providing a particular header address, allowing halting transfer of a data burst.

There is no technical relationship among the inventions listed above involving one or more of the same or corresponding special technical features in the sense of Rule 13.2 PCT.

Re Item V

Reasoned statement under Article 35(2) with regard to novelty, inventive step or industrial applicability; citations and explanations supporting such statement

Concerning claims 1-29 and 132

1. The broad wording of claim 1 includes synchronising methods such as known from document SHAW R H: "A COMPLETE GUIDE TO OS/2 INTERPROCESS COMMUNICATIONS AND DEVICE MONITORS" MICROSOFT SYSTEMS JOURNAL,US,MICROSOFT CO., REDMOND, WA, vol. 4, no. 5, 1 September 1989 (1989-09-01), pages 35-60, XP000568106 ISSN: 0889-9932 or MOGUL J C ET AL: "THE PACKET FILTER: AN EFFICIENT MECHANISM FOR USER-LEVEL NETWORK CODE" OPERATING SYSTEMS REVIEW (SIGOPS),US,ACM HEADQUARTER. NEW YORK, 1987, pages 39-51, XP002913603.
2. The additional features of claims 2-5 are also known from the R H Shaw document mentioned in the previous paragraph.
3. The combination of the features of dependent claim 6 is neither known from, nor

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

rendered obvious by, the available prior art. Claim 6 therefore meets the requirements of Article 33(2) and (3) PCT.

4. Dependent claims 7-26, so far as including claim 6, also meet the requirements of Article 33(2) and (3) PCT.
4. Independent claims 27-29 refer to claim 1. Independent apparatus claim 132 corresponds to method claim 1. Therefore, all observations in paragraphs 1-3 above apply correspondingly to these independent claims.

Concerning claims 72-79

1. The use of triggering values for arranging DMA data transfers as set out in independent method claim 72 is not obvious nor known from the prior art. Claim 72 therefore meets the requirements of Article 33(2) and (3) PCT.
2. Claims 72-79 are dependent from claim 2 and therefore also meet the requirement of Article 33(2) and (3) PCT.

Concerning claims 114-125

1. All features of claim 114 are known from document EP 0 784 268 A (see Abstract; section "System Architecture" on pages 4 and 5). The claim therefore lacks novelty.
2. The additional features of claim 115 refer to unexamined claims 80-99 (see Item III above), which relate to another invention.
3. The additional features of dependent claims 116-125 are either known from, or rendered obvious by, the prior art documents EP 0 784 268 A and US 5 787 251 A.

Concerning claims 126, 127, 128 and 129-131

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

The features of claims 126 and 128 that are adequately defined (see Item IIIV below) and of claim 129 appear to be known from document EP 0 772 368 A.

Re Item VIII

Certain observations on the international application

Concerning claims 1-29 and 132

1. Claims 1-5 are not supported by the description as required by Article 6 PCT, as their scope is broader than justified by the description and drawings.
2. The term "end-point application" used in claim 1 does not appear to have a well-recognised meaning in the art. The term should therefore have been clarified.

Concerning claims 126, 127, 128 and 129-131

Claims 126 and 128 do not meet the requirements of Article 6 PCT in that the matter for which protection is sought is not clearly defined. The claims attempt to define the subject-matter in terms of the result to be achieved (the destination addresses being inferable in claim 126 and calculation of the addresses of each data word in claim 128), which merely amounts to a statement of the underlying problem. The technical features necessary for achieving this result should have been added (see, for example, description page 39, second paragraph)..

Concerning all claims

1. The Independent claims are not in the two-part form in accordance with Rule 6.3(b) PCT, which in the present case would be appropriate, with those features known in combination from the prior art being placed in the preamble (Rule 6.3(b)(i) PCT) and with the remaining features being included in the characterising part (Rule 6.3(b)(ii) PCT).

**INTERNATIONAL PRELIMINARY
EXAMINATION REPORT - SEPARATE SHEET**

International application No. PCT/GB00/01691

2. The features of the claims are not provided with reference signs placed in parentheses (Rule 6.2(b) PCT).

This Page Blank (uspto)

PCT

INTERNATIONAL SEARCH REPORT

(PCT Article 18 and Rules 43 and 44)

Applicant's or agent's file reference JSR.P51122PC	FOR FURTHER ACTION see Notification of Transmittal of International Search Report (Form PCT/ISA/220) as well as, where applicable, item 5 t.	
International application No. PCT/GB 00/ 01691	International filing date (day/month/year) 03/05/2000	(Earliest) Priority Date (day/month/year) 04/05/1999
Applicant AT&T LABORATORIES-CAMBRIDGE LIMITED et al.		

This International Search Report has been prepared by this International Searching Authority and is transmitted to the applicant according to Article 18. A copy is being transmitted to the International Bureau.

This International Search Report consists of a total of 6 sheets.

☒ It is also accompanied by a copy of each prior art document cited in this report.

1. Basis of the report

- a. With regard to the language, the international search was carried out on the basis of the international application in the language in which it was filed, unless otherwise indicated under this item.
- ☐ the international search was carried out on the basis of a translation of the international application furnished to this Authority (Rule 23.1(b)).
- b. With regard to any nucleotide and/or amino acid sequence disclosed in the international application, the international search was carried out on the basis of the sequence listing :
- ☐ contained in the international application in written form.
- ☐ filed together with the international application in computer readable form.
- ☐ furnished subsequently to this Authority in written form.
- ☐ furnished subsequently to this Authority in computer readable form.
- ☐ the statement that the subsequently furnished written sequence listing does not go beyond the disclosure in the international application as filed has been furnished.
- ☐ the statement that the information recorded in computer readable form is identical to the written sequence listing has been furnished.
2. ☐ Certain claims were found unsearchable (See Box I).
3. ☒ Unity of invention is lacking (see Box II).

4. With regard to the title,

- ☒ the text is approved as submitted by the applicant.
- ☐ the text has been established by this Authority to read as follows:

5. With regard to the abstract,

- ☒ the text is approved as submitted by the applicant.
- ☐ the text has been established, according to Rule 38.2(b), by this Authority as it appears in Box III. The applicant may, within one month from the date of mailing of this international search report, submit comments to this Authority.

6. The figure of the drawings to be published with the abstract is Figure No.

- ☒ as suggested by the applicant.
- ☐ because the applicant failed to suggest a figure.
- ☐ because this figure better characterizes the invention.
- 10
☐ None of the figures.

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/46 H04L29/06

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EP0-Internal, IBM-TDB, INSPEC, COMPENDEX, PAJ

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	SHAW R H: "A COMPLETE GUIDE TO OS/2 INTERPROCESS COMMUNICATIONS AND DEVICE MONITORS" MICROSOFT SYSTEMS JOURNAL, US, MICROSOFT CO., REDMOND, WA, vol. 4, no. 5, 1 September 1989 (1989-09-01), pages 35-60, XP000568106 ISSN: 0889-9932	1-5, 7-29, 132
A	page 52, right-hand column, paragraph 2 -page 56, right-hand column, paragraph 3 — -/-	6

☒ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

* Special categories of cited documents:

- *A* document defining the general state of the art which is not considered to be of particular relevance
- *E* earlier document but published on or after the international filing date
- *L* document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
- *O* document referring to an oral disclosure, use, exhibition or other means
- *P* document published prior to the international filing date but later than the priority date claimed

T later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

X document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

Y document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

& document member of the same patent family

Date of the actual completion of the international search

30 March 2001

Date of mailing of the international search report

09.04.2001

Name and mailing address of the ISA

European Patent Office, P.B. 5818 Patentaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040, Tx. 31 651 epo nl,
Fax: (+31-70) 340-3016

Authorized officer

Beltrán-Escavy, J

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT		
Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	MOGUL J C ET AL: "THE PACKET FILTER: AN EFFICIENT MECHANISM FOR USER-LEVEL NETWORK CODE"	1,132
A	OPERATING SYSTEMS REVIEW (SIGOPS),US,ACM HEADQUARTER. NEW YORK, 1987, pages 39-51, XP002913603 abstract	2-71, 80-113, 133-139
A	page 39, right-hand column, paragraph 3 page 40, right-hand column, paragraph 2 - paragraph 3 page 41, right-hand column, paragraph 3 -page 42, left-hand column, paragraph 1 page 45, left-hand column, paragraph 4 page 46, right-hand column, paragraph 2 - paragraph 6	
A	US 5 438 640 A (SASAKA EISUKE ET AL) 1 August 1995 (1995-08-01) abstract column 1, line 1 -column 3, line 21 column 15, line 61 -column 16, line 58	1-29,132
A	US 5 586 273 A (BLAIR DANA L ET AL) 17 December 1996 (1996-12-17) abstract column 1, line 1 -column 4, line 30 column 6, line 52 -column 7, line 49 figures 5A-5E,6A-6E claim 6	1-71, 80-113, 132-139
A	EP 0 251 584 A (NORTHERN TELECOM LTD) 7 January 1988 (1988-01-07) abstract column 1, line 1 -column 7, line 45 column 9, line 5 -column 10, line 13	1,30,45, 55,80, 100,132, 133
A	ANONYMOUS: "Low-Level Interprocess Communication Facility Interface" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 30, no. 10, 1 March 1988 (1988-03-01), pages 146-148, XP002151749 New York, US the whole document	1,132

	-/-	

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EP 0 359 137 A (NAT SEMICONDUCTOR CORP) 21 March 1990 (1990-03-21) cited in the application abstract page 1, line 1 -page 4, line 7 claims 4-16,24-32	1,132
A	PATENT ABSTRACTS OF JAPAN vol. 010, no. 074 (P-439), 25 March 1986 (1986-03-25) & JP 60 211559 A (NIPPON DENKI KK), 23 October 1985 (1985-10-23) cited in the application abstract	1,132
A	US 5 727 154 A (CHRISTOPHER CHRISTOPHER D ET AL) 10 March 1998 (1998-03-10) the whole document	30-71, 80-113, 133-139
A	GB 2 315 638 A (ERICSSON TELEFON AB L M) 4 February 1998 (1998-02-04) the whole document	30-71, 80-113, 133-139
A	US 4 429 387 A (KAMINSKI STANLEY X) 31 January 1984 (1984-01-31) abstract column 1, line 1 -column 6, line 45	72-79
X	EP 0 784 268 A (SUN MICROSYSTEMS INC) 16 July 1997 (1997-07-16) abstract page 1, line 1 -page 9, line 4	114-125
X	US 5 787 251 A (HAMILTON GRAHAM ET AL) 28 July 1998 (1998-07-28) abstract column 1, line 1 -column 4, line 48	114-125
X	EP 0 772 368 A (SUN MICROSYSTEMS INC) 7 May 1997 (1997-05-07) the whole document	126-131
A	US 5 179 556 A (TURNER JONATHAN S) 12 January 1993 (1993-01-12) abstract column 1, line 1 -column 5, line 30 claims 1-23	126-131
A	US 5 778 175 A (PAUL GIDEON ET AL) 7 July 1998 (1998-07-07) abstract column 5, line 24 - line 54	126-131

INTERNATIONAL SEARCH REPORT

International application No.
PCT/GB 00/01691

Box I Observations where certain claims were found unsearchable (Continuation of item 1 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:
2. ☐ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:
3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box II Observations where unity of invention is lacking (Continuation of item 2 of first sheet)

This International Searching Authority found multiple inventions in this International application, as follows:

see additional sheet

1. ☒ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.
2. ☐ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.
3. ☐ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:
4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

- ☐ The additional search fees were accompanied by the applicant's protest.
- ☒ No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. Claims: 1-29, 132

Method for synchronising an end-point application in a computer.

2. Claims: 30-44, 45-54, 55-71, 80-99, 100-113, 133-139

Method of synchronising between a sending application in a first computer and a second application in a second computer.

3. Claims: 72-79

Method of arranging data transfers on a computer, by means of a Direct Memory Access (DMA) engine.

4. Claims: 114-125

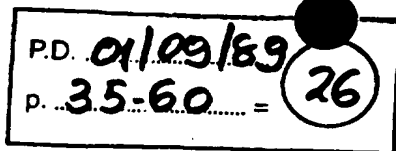
Method of sending a request from a client application on a first computer to a server application on a second computer, and of sending a response from the server application to the client application.

5. Claims: 126-127, 128, 129-131

Method of arranging data for transfer as a data burst over a computer network.

Patent document cited in search report		Publication date	Patent family member(s)	Publication date
US 5438640	A	01-08-1995	JP 6174954 A AU 666257 B AU 5936794 A CA 2120792 A CN 1103719 A, B EP 0634677 A KR 175074 B	24-06-1994 01-02-1996 27-01-1995 17-01-1995 14-06-1995 18-01-1995 01-05-1999
US 5586273	A	17-12-1996	US 5859984 A	12-01-1999
EP 0251584	A	07-01-1988	CA 1253971 A JP 63040959 A	09-05-1989 22-02-1988
EP 0359137	A	21-03-1990	US 5140679 A DE 68925322 D DE 68925322 T DE 68929099 D DE 68929099 T EP 0632391 A EP 0634722 A JP 2257242 A KR 129000 B US 5199105 A US 5228130 A	18-08-1992 15-02-1996 05-09-1996 16-12-1999 20-07-2000 04-01-1995 18-01-1995 18-10-1990 15-04-1998 30-03-1993 13-07-1993
JP 60211559	A	23-10-1985	NONE	
US 5727154	A	10-03-1998	NONE	
GB 2315638	A	04-02-1998	NONE	
US 4429387	A	31-01-1984	DE 3363855 D EP 0085971 A	10-07-1986 17-08-1983
EP 0784268	A	16-07-1997	US 5887172 A JP 2978807 B JP 9305419 A JP 11353199 A	23-03-1999 15-11-1999 28-11-1997 24-12-1999
US 5787251	A	28-07-1998	US 5577251 A DE 69327448 D EP 0604010 A JP 6231069 A US 5566302 A US 6157961 A	19-11-1996 03-02-2000 29-06-1994 19-08-1994 15-10-1996 05-12-2000
EP 0772368	A	07-05-1997	US 5758089 A JP 9266485 A	26-05-1998 07-10-1997
US 5179556	A	12-01-1993	AU 2425792 A WO 9303555 A	02-03-1993 18-02-1993
US 5778175	A	07-07-1998	EP 0811285 A JP 11501196 T WO 9723976 A	10-12-1997 26-01-1999 03-07-1997

This Page Blank (uspto)



A Complete Guide to OS/2 Interprocess Communications and Device Monitors

35

Richard Hale Shaw

The protected mode operating environment in the OS/2 operating system allows each program to run in a different session independently of other programs. Yet this same protection mechanism prevents a program from readily sharing data and communicating with other processes. Fortunately, OS/2¹ offers several interprocess communications facilities (IPC) that allow two processes to signal, synchronize, serialize resources, and transfer data. This article explores OS/2 IPC and introduces OS/2 character device monitors.

OS/2 has come a long way. When I wrote what would become the first article in this series, most OS/2 developers were still using OS/2 Version 1.0 or beta copies of Version 1.1 from the Microsoft® OS/2 Software Development Kit. With the addition of Microsoft Word Version 5 for OS/2, I have written, developed, and edited this article and its sample programs entirely under OS/2—something I have not been able to do in the past. I can run each tool—word processor, program editor, compiler, test programs—in different Presentation Manager windows, switching between them with a mouse click. This capability is another example of how productive an environment OS/2 can be.

In previous articles, I discussed OS/2's protected mode operating environment and how it is organized in screen groups which are composed of one or more sessions. A session and a screen group are usually synonymous. The exception is the Presentation Manager's screen group, which can run more than one session simultaneously. Each session consists of one or more processes (roughly synonymous with programs), each of which has one or more threads of execution. Since each session is independent of the others, a process can run without concern that it will be unnecessarily interrupted by others or that another process will wantonly hang the system.

However, this same protection and independence makes it difficult for one process to communicate directly with another one. OS/2 isolates the data of one program from the data of another program. You can't just cast a pointer to a common data area as you would under a real-mode environment like the MS-DOS[®] operating system, therefore sharing data is more complex under OS/2.

Fortunately, OS/2 offers a rich set of interprocess communications (IPC) facilities that allow for flexible communications between well-behaved, cooperating programs. In this article, I'll

OS/2 ISOLATES THE DATA
OF ONE PROGRAM FROM
THE DATA OF ANOTHER
PROGRAM. YOU CAN'T JUST
CAST A POINTER TO A
COMMON DATA AREA AS YOU
WOULD UNDER A REAL-
MODE ENVIRONMENT LIKE
THE MS-DOS OPERATING
SYSTEM, THEREFORE
SHARING DATA IS MORE
COMPLEX UNDER OS/2.

Richard Hale Shaw, a writer who contributes to various computer magazines, is the author of an upcoming book on OS/2 programming.

SEPTEMBER 1989

**QUEUES ARE A
POWERFUL FORM OF
IPC UNDER OS/2;
THEY REQUIRE MORE
WORK FROM YOUR
PROGRAM, BUT THEY
ALLOW SEVERAL
SENDER PROCESSES
TO TRANSFER DATA
TO A SINGLE
RECEIVER PROCESS.**

discuss the forms of IPC, then each of the IPC facilities provided by OS/2. I'll also present a sample program for each IPC that will show you how to use them in an application program. And I'll introduce you to OS/2 character device monitors, which allow an application to intercept data from a device driver long before it reaches another application.

Under OS/2, you can create pop-up programs with keyboard monitors, a subject that I will also cover.

An OS/2 IPC Overview

Clearly, there are times when it is advantageous for threads in different processes to communicate. For instance, a process may need to signal another that an event has occurred, or it may need to synchronize its activities with those of another process. At other times, several processes may need to share a resource that only one process at a time may use, or a process may need to transfer data to another process.

The most primitive form of IPC is signaling, which is used when a thread in one process needs to notify a thread in another process that an event has occurred. Although you should use semaphores for signaling most of the time, you can also use the OS/2 signaling facility to pass signals between processes. OS/2 signals are crude, but they do allow a process to notify another of an event. Signals are also used to notify an application that the user is trying to terminate the process by typing Control-C or Control-Break.

Synchronization, which is

required when a thread needs to coordinate its activities with those of another thread, is another form of IPC. Semaphores are the simplest and easiest OS/2 tools to use for this purpose, but you can also use queues and shared memory.

Yet another form of IPC is serialization, which allows one thread to ensure that it has exclusive use of, or owns, a resource that only one process can access or use at a time. Semaphores are usually the perfect solution to providing exclusivity as long as every process that wants to use the resource must first gain control of the semaphore.

RAM semaphores are used to signal, serialize resources, and synchronize threads in the same process. For interprocess signaling, serialization, and synchronization, you'll want to use OS/2 system semaphores most of the time. (OS/2 also offers Fast-Safe RAM semaphores, which are usually used with dynamic-link libraries.)

Interprocess data transfer is another form of IPC. It is used when a thread in one process needs to transfer data to a thread in another process. Of course, data transfer isn't a problem between threads in the same process, since they can access the same global data. OS/2's shared memory, pipes, and queues are available to facilitate interprocess data transfer. Pipes are the easiest to implement and use if the data transfer is going to take place between two related processes (that is, where one process is the parent of the other). Shared memory is slightly more involved, since the application must perform a modest amount of memory management; but it allows you to transfer data between two programs as if each had the same variables in its global data area. Queues are a powerful form of

IPC under OS/2; they require more work from your program, but they allow several sender processes to transfer data to a single receiver process. Named pipes, the most powerful form of OS/2 IPC, are almost limitless in their use—even across a network. I've given the discussion of named pipes special treatment in the sidebar "Named Pipes: A Welcome Latecomer."

System Semaphores

A RAM semaphore is a 4-byte long variable. You can define a semaphore external to any function in the application and make it globally available to all threads in the process. There are no limits to the number of RAM semaphores you can use in an application. Unfortunately, only the threads in the process that defines a RAM semaphore can access it. While you can place a RAM semaphore in shared memory and allow threads in different processes to access it there, you needn't bother: use system semaphores instead.

System semaphores, used for signaling, synchronization, or serialization, are the most versatile IPC mechanism that OS/2 offers. Like their RAM semaphore cousins, system semaphores have two distinct states, set and clear. You can use the same application program interface (API) services on system semaphores that you use to manipulate RAM semaphores. Using these services ensures that the action of setting or clearing a system semaphore is a single atomic OS/2 instruction that the OS/2 system scheduler cannot preempt.

A thread must always reference a semaphore by means of a semaphore handle. The address of a RAM semaphore is its semaphore handle, so all you have to do is define it and initialize it to 0L to use it. System semaphores must be created

with the `DosCreateSem` function or opened with the `DosOpenSem` function before they can be used. Then OS/2 will return a handle for the semaphore. Only one process can create a particular system semaphore, but any other process can open the semaphore once it has been created. Thus every process can obtain the semaphore's handle.

The naming convention that OS/2 uses for system semaphores is essentially the same one used for shared memory and queues. It requires a pseudo-filename which, in the case of system semaphores, is preceded by a pseudo-directory name, `\SEM`. For instance, a thread can create a system semaphore called `\SEMSEM0`. The `\SEM` directory doesn't really exist; it's a means of identifying `SEM0` as a system semaphore. To create a system semaphore, a thread must pass the semaphore's name to the `DosCreateSem` function, which returns the semaphore handle. The creating thread must specify whether other processes will be able to change or only monitor the semaphore. Then other processes can call `DosOpenSem`, which will open the semaphore and return a handle for them to use. Note that OS/2 Version 1.1 allows a maximum of 256 system semaphores at any time, so be judicious in their use.

When a thread is finished with a system semaphore, it should call `DosCloseSem` to close it. If the thread owns the system semaphore at the time, it should always clear the semaphore before closing it. Closing the semaphore tells OS/2 that the thread is no longer using it. OS/2 will return an error if the thread attempts to use a system semaphore after closing it, even if the thread was the one that created the semaphore. Once the

last thread closes it, OS/2 will discard the semaphore.

As already mentioned, a thread can use a system semaphore handle with any of the API functions used for manipulating RAM semaphores. Of those functions, `DosSemClear` and `DosSemSet` are best for signaling, synchronization, and mutual exclusion. `DosSemRequest` will cause the calling thread to wait, or block, while another thread owns the semaphore. Once its owner clears the semaphore with `DosSemClear`, `DosSemRequest` will set the semaphore and the blocked thread will unblock and return.

There are three additional functions that you can use to make a thread wait for a change in a semaphore's state. These functions allow a thread to synchronize its actions with the actions of another thread. `DosSemWait` will wait for a thread to call `DosSemClear` and clear a semaphore (like `DosSemRequest` does except `DosSemWait` doesn't set the semaphore). `DosSemSetWait` does the same thing but sets the semaphore before it begins to block. `DosMuxSemWait` lets you construct a list of semaphores and returns when any semaphore in the list has been cleared.

Sample Semaphore

`SSEM.C` (see Figure 1) demonstrates the use of system semaphores. It shows how to create or open a system semaphore and how to use it with `DosSemWait`, `DosSemRequest`, `DosSemSet`, and `DosSemClear`. It also illustrates a simple example of synchronization between processes.

What's unique about `SSEM` is that to use it, you should run two or more instances of it in separate sessions. The first instance will always create the system semaphore, set it, and enter a

program loop. Each time the program goes through the loop, it prints a message and clears the semaphore. Then it looks to see if the operator has pressed a key and, if so, closes the semaphore and terminates itself. If the operator has not pressed a key, the process sleeps for a while and requests ownership of the semaphore. When the semaphore becomes available, `SSEM` begins the loop again.

A subsequent instance of `SSEM`, unable to create the semaphore because it already exists, will open the semaphore instead. Then it will wait until some other instance of `SSEM` clears the semaphore and enters the loop described above. Thus multiple instances of the same program will take turns owning the semaphore.

The program doesn't really use the semaphore to protect a resource, but using the semaphore does force the program to indirectly serialize its use of the OS/2 video resources. Only one instance of the program can write its message at a time—when the semaphore is available. This sample should give you a better idea of how system semaphores work.

Anonymous Pipes

UNIX system developers originally introduced pipes as a form of operating system IPC. Pipes get their name from their resemblance to a water pipe, in which water transfers in one direction from one location to another. In an operating system context, a pipe allows a process to send data to another process, as if the first is writing to a file that the second is reading. You can also use pipes for redirection, since, in many cases, `STDIN` and `STDOUT` are indistinguishable from file handles. OS/2 anonymous pipes are a very efficient, versatile means of interprocess data transfer and

Figure 1: Source Code for System Semaphores

SSEM.MAK

```
# make file for ssem.c
#
COPT=/Lp /W3 /Zp1 /G2s /IS(INCLUDE) /Alfw

ssem.exe: ssem.c ssem.mak
cl $(COPT) ssem.c /link /co /noi libbcm
markexe windowcompat ssem.exe
```

SSEM.C

```
/* ssem.c RRS 5/1/89
This program demonstrates the use of system semaphores. To use it,
run more than one copy of the program in different sessions with:

SSEM
*/
#define INCL_DOS
#define INCL_KBD
#define INCL_ERRORS

#include<os2.h>
#include<stdio.h>

#define SEMNAME "\\sem\\ssem.sem"

void error_exit(USHORT err, char *msg);
PID Os2GetPid(void);
void main(void);

void main(void)
{
    BSYSSEM semhandle;
    USHORT retval;
    int creator = FALSE;
    KBDKEYINFO kbdkeyinfo;
    PID pid;
    unsigned count = 0;

    pid = Os2GetPid();

    /* first process will successfully execute this code and create the
    semaphore */
    if(retval = DosCreateSem(CSEM_PUBLIC, &semhandle, SEMNAME))
    {
        if(retval != ERROR_ALREADY_EXISTS)
            error_exit(retval, "DosCreateSem");

        /* second process will open the semaphore previously created */
        else if (retval = DosOpenSem(&semhandle, SEMNAME))
            error_exit(retval, "DosOpenSem");

        DosSemWait(semhandle, SEM_INDEFINITE_WAIT);
        DosSemRequest(semhandle, SEM_INDEFINITE_WAIT);
    }
    else
    {
        /* then first process will wait here until second process executes the
        code in the loop below */
        DosSemSet(semhandle);
        creator = TRUE;
    }

    while(TRUE) /* both processes will continue to execute in
    this loop */
    {
        printf("%s (%u) %u\n", (creator ? "Creating Thread" :
        "User Thread"), pid, count++);

        DosSemClear(semhandle);

        KbdCharIn(&kbdkeyinfo, IO_NOWAIT, 0);
        /* check for key press */
        if(kbdkeyinfo.fbStatus & FINAL_CHAR_IN)
            /* if pressed, break out */
            break;
    }
}
```

demand little from the programmer or the application.

While anonymous pipes are flexible to use, their application is limited: you can only use them between related processes, where one process is the parent (or an ancestor) of another. Thus one process can create an anonymous pipe and start or spawn another process that can read from it or write to it. I will discuss how to use anonymous pipes, but move on rather quickly to more powerful forms for transferring data in IPC.

A pipe is a first in, first out (FIFO) stream in which one process inserts data at one end and another takes the data out at the other end. It resembles a file to the processes that use it but it is actually a memory buffer that OS/2 manages. A process can write data to a pipe and another process can read data from the pipe as if either were manipulating a disk file. Indeed, several processes can write data to the pipe, but only one process can read from it. OS/2 manages the pipe and handles the scheduling and synchronization of the data.

Under OS/2, pipe handles, like file handles, can be inherited by a child process. Thus, a parent process can pass a pipe handle to a child process as a command line parameter or via shared memory and the child can use the handle.

The parent process can also use a pipe to read the child's standard output or write to its standard output. This is done by closing either the standard input or output handle and substituting a pipe handle in its place. When the child uses either standard input or standard output, it will actually be using the pipe handle. For instance, a parent process can call `DosClose` to close `STDOUT`. By calling `DosDupHandle`, the parent process creates a duplicate of the pipe's write handle and forces it

to be recognized as STDOUT. When the child process begins, all output that it writes to STDOUT will actually go to the pipe where the parent process can read it. This will become clearer to you in the PIPE sample program.

To create an anonymous pipe, a process must call `DosMakePipe`. This API function returns two handles to the pipe, one for reading and one for writing. The receiver can pass one handle to a child process as a command line parameter or by means of shared memory (see the section on shared memory below). Then one process can use `DosRead` to read from the pipe and the other can use `DosWrite` to write to it. `DosWrite` will not return until the data passed to it has been written to the pipe, so it may have to wait until a `DosRead` by the receiver has made enough room in the pipe for it to complete its task and return.

There are a few additional differences between pipes and files other than that pipes do not involve disk I/O: for one, a process cannot seek in a pipe, but it doesn't have to maintain a file pointer. For another, an application needn't worry about whether the data in the pipe will be overwritten, since OS/2 manages the pipe. Also, a pipe is inherently faster than a disk file since OS/2 stores the entire pipe in memory. On the other hand, although a file is limited only by the available disk space, a pipe can never be larger than 64Kb: the creating process can suggest a pipe size to OS/2 when it calls `DosMakePipe`, but OS/2 may dynamically change the size of the pipe as it is used.

An application doesn't have to worry about filling up the pipe unless data is being read from the pipe too slowly or written to the pipe too quickly. Alternatively, a thread can use

Figure 1

```

        DosSleep(150L);
        DosSemRequest(semhandle, SEM_INDEFINITE_WAIT);
    }

    DosCloseSem(semhandle); /* close the semaphore */
    DosExit(EXIT_PROCESS, 0); /* and exit */
}

PID Os2GetPid(void)          /* returns process id */
{
    PIDINFO pidinfo;

    DosGetPID(&pidinfo);     /* get process id */
    return pidinfo.pid;       /* return it */
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

```

`DosWriteAsync` and `DosReadAsync` so that it won't be blocked while the pipe is full or being written to. But these functions create their own threads and require a huge stack to accommodate any combination of API calls. It's more reasonable to create your own thread that performs these operations asynchronously using calls to `DosWrite` or `DosRead` rather than using `DosWriteAsync` and `DosReadAsync`.

Since data is written to a pipe in FIFO order, it is up to the processes involved to synchronize themselves. If the data written to the pipe varies in length, your applications will have to work out some sort of protocol, or procedure for knowing how to interpret the data. Once an application finishes using the pipe, it should close it with `DosClose`. When all the processes using the pipe have closed their pipe handles, OS/2 will delete the resources used to manage the pipe.

Sample Pipe Program

PIPE.C (see Figure 2) illustrates a simple use of an anonymous pipe. The program also demonstrates a technique that an application might use when it wants to start another process in the same session and give the

operator the ability to monitor the new process's output. When you run PIPE, it divides the screen into two halves (upper and lower) and displays the child program's output in the upper half while interacting with the operator in the lower half. It does this by closing the STDOUT handle and substituting it with a pipe handle. Thus, the child process thinks it is writing to STDOUT, when its output is actually going to an anonymous pipe. The parent process uses a separate thread to read the pipe and display the output in the lower half of the screen.

To run PIPE, enter
PIPE <programname> [args...]
 and follow PIPE with the name of the program (and any arguments) whose output will be displayed in the upper half of the screen. PIPE begins by creating an anonymous pipe and closing STDOUT. Then it calls `DosDupHandle` to duplicate the pipe's write handle as the new STDOUT. Using the pipe handle as the new STDOUT does not hinder PIPE itself, since it uses VIO for its own video output. But if the child program uses VIO or STDERR, they will not be transferred to PIPE for printing on the screen.

Next, PIPE clears the screen

and divides it in two (by using `VioWrtNCell` to draw a line horizontally across the center of the screen). Then it creates a buffer that includes both the program to run and its arguments and calls `DosExecPgm` to run the child process asynchronously, allowing the child and PIPE to run at the same time in the same session.

After this, PIPE creates a new thread of execution to read the pipe input and print it on the screen. Contained in the function `readchild`, this thread enters a program loop where it uses a call to `DosRead` to read from the pipe. When `DosRead` returns, the thread strips the pipe input of carriage returns and line feeds and scrolls the upper portion of the screen up as it prints each line. If it finds any incomplete lines, it moves them to the beginning of the buffer, sleeps briefly, and returns to the `DosRead` at the top of the loop.

Simultaneously, the main thread also enters a loop, where it calls `KbdStringIn` to read a line from the user. As the thread gets each line, it scrolls the lower half of the screen up, prints the line, and returns to the top of the loop. If the line typed was `QUIT`, the thread breaks out of the loop. Then it calls `DosCwait` to see if the child process has completed. If not, it calls `DosKillProcess` to terminate the child process and terminates itself.

You can use PIPE with any program that writes its output to `STDOUT`. I've written a simple test program, `TEST.C`, for testing the PIPE program. `TEST.C`, which can be downloaded from any of the *MSJ* bulletin boards, prints a list of messages to `STDOUT` in an infinite loop.

Shared Memory

Shared memory is an OS/2 IPC mechanism that allows more than one process to use the

same memory segment at the same time. It's useful for inter-process data sharing and data transfer, since one process can transfer data to another by copying it to a shared memory segment. Then another process can read it and modify it, and the original process can read it back. An unlimited number of processes can access shared memory, all of whom can read from or write to the shared memory segment.

Other than a small amount of code required to set up shared memory among processes, a program can access a shared memory segment by means of a far pointer, making shared memory access relatively transparent to an application. This flexibility usually demands some method for establishing mutual exclusion; otherwise one process may overwrite the data of another. Because of this you should usually plan on using a system semaphore to manage each shared memory segment.

There are three ways of establishing shared memory. In each one, a process—the owner of the shared memory—allocates a shared memory segment that a second process will be able to access. Using the first method, the owner process gives the second process access to the shared memory by creating a memory selector that the second process can use. This is known as givable memory; it requires that the owner know the process ID of the second process in order to create the new selector. Under the second approach, known as gettable memory, the owner passes its own memory selector to the second process, which can then get its own selector and access the memory segment.

An alternative technique involves named shared memory. Under this scheme, a process can create a shared memory segment that is associated with a

specific name. Then any process that knows the name of the shared memory can access it directly.

Of the three approaches to shared memory, I think named shared memory is the most convenient and flexible to use. However, the first two methods are terrific for limiting shared memory access to a few specific processes and work particularly well with queues.

To use unnamed shared memory, a process must first call `DosAllocSeg` and allocate the shared segment, specifying the segment size and whether the segment is shareable. `DosAllocSeg` will return a selector to the segment. OS/2 relates selectors and processes, so that one process cannot use another's memory selector. Thus if the process knows the process ID of the second process, it can create givable shared memory by passing the selector and ID to `DosGiveSeg`, which will return a new selector the other process can use. Finally, the process must pass this selector to the other process, as a command line parameter or using some other mechanism. Queues lend themselves perfectly to this scheme, since you can create a pointer from the shared memory selector and pass the pointer through the queue. Or the first process can pass the original selector to the other process, which can use `DosGetSeg` to access the unnamed shared memory.

If your application doesn't know the ID of the other process, then you will want to use named shared memory. With named shared memory, your application uses a name for the shared memory segment which resembles the pseudo-filename used by system semaphores. An example would be `\SHAREMEM\MEM`; the `SHAREMEM` pseudo-directory

name indicates that a shared memory segment is being created. The application passes this shared memory name to DosAllocShrSeg, which returns a selector to the shared segment. As with system semaphores, any other process can open a shared memory segment once it has been created. The other processes must call DosGetShrSeg, which will return a selector to the named shared segment.

Under either scheme, a process should call DosFreeSeg to free the shared memory once it has finished with it. OS/2 will not discard the segment until the last process using it has called DosFreeSeg. As with system semaphores, a process can continue to access a shared segment even after the segment is free and the process that created the segment has terminated.

Once a process has obtained the selector to a shared memory segment, it can convert the selector into a far pointer using the MAKEP macro and access the shared memory directly.

Shared Memory Program

SHMEM.C (see Figure 3), a sample shared memory program, uses named shared memory. There is no need for a program for unnamed shared memory, since the sample program for queues serves adequately for both.

As with the sample system semaphore program, you can run multiple copies of the shared memory program. Each instance of the program competes for access to a piece of named shared memory by using a system semaphore. Once an instance of the program gains access to this memory, it looks to see if another instance of the program has placed a message there and, if so, prints it. Then it writes its own message to the shared memory segment, releases the semaphore (and

Figure 2: Source Code for Pipe Example

PIPE.MAK

```
# make file for pipe.c
#
COPT=/Lp /W3 /Zpie1 /G2s /I$(INCLUDE) /Alfw

pipe.exe: pipe.c pipe.mak
cl $(COPT) pipe.c /link /co /noi llibcmt
markexe windowcompat pipe.exe
```

PIPE.C

```
/* pipe.c RHS 5/2/89
pipe example:
this program will reroute the stdout of a child process to the
pipe, then it will print the child's output to the upper portion
of the screen while you can type into the lower portion of the
screen

To run:

PIPE <child program name> [child program arguments]
*/
#define INCL_DOS
#define INCL_KBD
#define INCL_VIO
#define INCL_ERRORS

#include<os2.h>
#include<nt\stdio.h>
#include<nt\process.h>
#include<nt\string.h>

#define PIPESIZE 80
#define STDIN 0
#define STDOUT 1
#define FAULBUFLN 128
#define ENV 01
#define THREADSTACK 800

char readbuf[PIPESIZE+5];
HFILE readhandle, writehandle, newSTDOUT = STDOUT;
USHORT cell = 0x1f20;
char readchild_stack[THREADSTACK];

void error_exit(USHORT err, char *msg);
PID Os2GetPid(void);
void main(int argc, char **argv);
void readchild(void);

void main(int argc, char **argv)
{
    USHORT retval, cell2 = 0x1fc4, i;
    STRINGINBUF stringinbuf;
    char failbuf[FAULBUFLN];
    RESULTCODES resultcodes;
    char keybuf[PIPESIZE], *p;
    PID pid, pidaesync;

    if(argc < 2) /* must have one argument */
        error_exit(0, "main");

    /* create pipe */
    if(retval = DosMakePipe(&readhandle, &writehandle, PIPESIZE))
        error_exit(retval, "DosMakePipe");

    DosClose(STDOUT); /* close stdout */

    /* pipe write handle now stdout */
    if(retval = DosDupHandle(writehandle, &newSTDOUT))
        error_exit(retval, "DosDupHandle");

    /* clear the screen */
    VioScrollUp(0, 0, 0xffff, 0xffff, 0xffff, (PBYTE)&cell, 0);
    VioWriteCell((PBYTE)&cell2, 80, 11, 0, 0);
    /* draw line to split screen */
```

Figure 2

```

memset(readbuf, 0, sizeof(readbuf)); /* clear buffer */
strcpy(readbuf, argv[1]); /* add program name */
if (!strcmp(strupr(readbuf), ".EXE")) /* tack on .EXE */
    strcat(readbuf, ".EXE");

p = (&readbuf[strlen(readbuf))+1);
for (i = 2; i < argc; i++) /* add program arguments */
{
    strcat(p, argv[i]);
    strcat(p, " ");
}

/* run the child program */
if (retval = DosExecPgm(failbuf, FAILBUFLen, EXEC_ASYNC,
    readbuf, ENV, &resultcodes, readbuf))
    error_exit(retval, "DosExecPgm");
pidasync = resultcodes.codeTerminate;

/* start pipe read thread */
if (_beginthread(readchild, readchild_stack,
    THREADSTACK, NULL) == -1)
    error_exit(-1, "_beginthread");

while (TRUE)
{
    stringinbuf.cb = sizeof(keybuf)-1;

    /* await operator input */
    if (retval = KbdStringIn(keybuf, &stringinbuf, IO_WAIT, 0))
        error_exit(retval, "KbdStringIn");

    keybuf[stringinbuf.cchIn] = '\0';

    /* scroll screen up */
    VioScrollUp(12, 0, 23, 0xffff, 1, (PBYTE)&cell, 0);

    /* write command */
    VioWrtCharStr(keybuf, strlen(keybuf), 23, 0, 0);

    /* clear prompt line */
    VioWrtNCell((PBYTE)&cell, 80, 24, 0, 0);
    if (!strcmp(strupr(keybuf), "QUIT"))
        /* if QUIT command, get out */
        break;
}

/* see if child finished */
if (retval = DosCwait(DCWA_PROCESSTREE, DCWN_NOWAIT,
    &resultcodes, &pid, resultcodes.codeTerminate))
{
    if (retval == ERROR_CHILD_NOT_COMPLETE) /*if not, kill it*/
    {
        if (retval = DosKillProcess(DKP_PROCESSTREE,
            pidasync))
            error_exit(retval, "DosKillProcess");
    }
    else if (retval != ERROR_WAIT_NO_CHILDREN)
        error_exit(retval, "DosCwait");
}

DosExit(EXIT_PROCESS, 0); /* get out */
}

void readchild(void) /* this thread reads the pipe, and
                     prints the lines */
{
    USHORT retval, bytesread, i, lines;
    char *p = readbuf;

    while (TRUE)
    {
        /* read the pipe */
        if (retval = DosRead(readhandle, p,
            PIPESIZE - (p - readbuf), &bytesread))

```

ownership of the shared memory), and blocks until it can gain access to the semaphore again. As with SSEM, an instance of SHMEM will terminate if you press a key while its session is in the foreground.

You can run more than one instance of SHMEM by running them in different sessions. Each instance prints messages left by other instances, but only one will create or own the shared memory. The messages identify the owner and other users of the shared memory, as well as their process IDs.

Queues

Queues are one of OS/2's most powerful and complex IPC mechanisms for data transfer. They are more flexible than anonymous pipes and potentially more powerful than shared memory. Queues are usually used in conjunction with shared memory and provide a framework that allows you to easily implement a shared memory manager. Used properly, they eliminate a lot of the work associated with shared memory data transfers.

Traditionally, queues are FIFO mechanisms in which one process inserts data at one end and another process takes it out at the other. In such cases, a process reading the queue must accept elements in the order in which they are written to it. OS/2 queues, on the other hand, can be ordered on a FIFO, last in, first out (LIFO), or priority basis. With priority order, the sender can assign a priority number to each element. The element with the highest priority is always first in the queue. If two elements have the same priority, they are placed in the queue in FIFO order. In addition, multiple processes can send data to the queue, whereas only one process—the queue owner—can read from it.

Unlike anonymous pipes, queues can be accessed by unrelated processes, although a process must know the name of the queue to access it. A process writes data to a queue as a distinct set of data elements. The sender, not the receiver, controls the size of the data it receives. You will typically use shared memory to transfer data via a queue: the queue transfers elements that contain information about the data being transferred. A sender will pass a pointer to the shared memory as part of a queue element. The receiver can then use this pointer to access the shared memory data.

A receiver or a queue owner has far more control over the queue elements than does the receiving process of a pipe. It can remove the queue elements in almost any order and can choose to leave some elements in the queue while retrieving others. In addition, the queue owner can decide to free the shared memory when it wishes or keep it around for a while. Unlike anonymous pipes, queues require their own set of API functions for sending and receiving data: you cannot use `DosRead` and `DosWrite` with queues. OS/2 limits queues to 409, with a total of 3268 data elements at a time.

Queues are relatively easy to use, but there are a few steps involved in accessing them. First, the receiving process (the queue owner) must create the queue with `DosCreateQueue`, specifying the name of the queue and the queue order. `DosCreateQueue` will then return a queue handle. The queue name is a pseudo-file-name like those used for system semaphores and named shared memory. An example is `QUEUESQNAME`. Once the queue owner creates the queue, other processes can open it via

Figure 2

```

        error_exit(retval, "DosRead");
        bytesread += (p-readbuf);

        for(i = lines = 0; i < bytesread; i++)
            /* remove CRs and LFs */
            {
                if(readbuf[i] == '\n')
                    lines++;
                if(readbuf[i] == '\r' || readbuf[i] == '\n')
                    readbuf[i] = '\0';
            }

        /* print each line found */
        for(p = readbuf, i = 0; i < bytesread && lines;
            lines--)
            {
                VioScrollUp(0, 0, 10, 0xffff, 1, (PBYTE)&cell, 0);
                VioWrtCharStr(p, strlen(p), 10, 0, 0);
                i += strlen(p);
                p += strlen(p);
                for( ; !(p) && i < bytesread; i++, p++);
                DosSleep(1L);
            }

        if(*p && !lines && i < bytesread)
            /* if anything is leftover */
            {
                memmove(readbuf, p, bytesread-i); /*move it up*/
                p = readbuf[bytesread-i];
            }
        else
            p = readbuf;
    }

PID Os2GetPid(void) /* returns process id */
{
    PIDINFO pidinfo;

    DosGetPID(&pidinfo); /* get process id */
    return pidinfo.pid; /* return it */
}

void error_exit(USHORT err, char *msg)
{
    char buf[100];

    sprintf(buf, "Error %u returned from %s\r\n", err, msg);
    VioWrtTTY(buf, strlen(buf), 0);
    DosExit(EXIT_PROCESS, 0);
}

```

`DosOpenQueue`, which will return a queue handle. This function also returns the process ID of the queue owner, so that a sender can create a shared memory selector for the queue owner.

When a process is finished with the queue, it should close it using `DosCloseQueue`. OS/2 will maintain the queue as a resource until the owner closes the queue, at which time it disposes of the queue and any elements remaining in it. Attempts by a process to write to the queue after that will fail. A sender can reopen a queue after

closing it as long as the queue owner has not closed the queue.

As I mentioned earlier, queue data elements are typically stored in shared memory segments. There are, however, several ways that an application can pass data through a queue. The most common approach is to use unnamed shared memory, with the sender allocating a segment every time it places an element in the queue (these are segments the size of a queue element, not 64Kb segments). The sender writes data to the segment, creates a selector to the shared

UNLIKE
ANONYMOUS PIPES,
QUEUES CAN BE
ACCESSED BY
UNRELATED
PROCESSES,
ALTHOUGH A
PROCESS MUST
KNOW THE NAME OF
THE QUEUE TO
ACCESS IT.

memory that can be used by the queue owner, and converts the selector into a pointer. Then it calls `DosWriteQueue`, passing it a priority value and the pointer to the shared memory. The sender can then free the shared memory segment if it no longer needs it, since OS/2 will not discard the segment until the queue owner frees it.

Alternatively, the application can allocate one shared memory segment and divide it into pieces that the sender and queue owner will reuse. With this approach, the sender passes a pointer to the portion of the shared memory segment where the data resides, rather than a pointer to the segment itself. Although it is more work for the programmer to create a scheme to manage portions of the shared segment, the resulting program should run faster, since the constant allocation and reallocation of memory found in the first scheme is absent.

There is, however, a way to pass data via a queue without using shared memory. If the sender is only transferring small pieces of data that are no more than 2 bytes in size (such as a series of tokens or even selectors), the application can forgo the use of shared memory entirely. The `DosWriteQueue` function includes a 2-byte request code that OS/2 reserves for use by the application. An application can use this code to pass 2-byte data elements through the queue to the receiving process without

implementing either of the two shared memory schemes outlined above.

Regardless of the method you use, the receiver must use `DosReadQueue` to read the elements from the queue. This function will return the process ID of the sender, the request code described in the previous paragraph, the length of the data element, a pointer to the data element, and its priority. `DosReadQueue` can wait until an element is in the queue or return immediately if no data elements are available. While the receiver should typically devote a thread to reading the queue, it can avoid this by providing a semaphore handle to `DosReadQueue` that OS/2 will clear when data elements are in the queue. The receiver can check this semaphore from time to time to see if a queue element has arrived while it was tending to other cases.

The receiver can use `DosPeekQueue` to examine elements in the queue without removing them, and both sender and receiver can call `DosQueryQueue` to determine the number of elements in the queue. In addition, the receiver can flush the queue with `DosPurgeQueue`. Of course, it's the receiver's responsibility to free any shared memory segments passed with each element.

Queue Programs

The sample queue takes the shared memory example a step farther, moving these applications into the world of client-server relationships. This is why I have written two programs, one for the client and one for the server. The server's job is simple: if it gets a message from a client process, it must print that message. The client, not much more complex than the server, periodically sends messages to the server. The processes com-

municate the messages using queues and shared memory.

If you study the listing for `QSERVER.C` (see Figure 4), you will see that the server begins by creating the queue. Then, it enters a program loop and blocks on a call to `DosReadQueue`. Once a queue element is available, the server returns from `DosReadQueue` and calls `DosQueryQueue` to get the count of the number of elements in the queue. Next it prints the message contained in shared memory, the element priority, the process ID of the sender, and the element count. Then the server frees the shared memory that the message occupies. If the operator has pressed a key in the server's session, it breaks out of the program loop, closes the queue and exits the program. Otherwise, it returns to the top of the loop and the `DosReadQueue` call.

`QCLIENT.C` contains the source code for the client process. The client begins by opening the queue. If the server hasn't yet created the queue, the client will sleep for a second and try again. As you can see, the order in which you run the programs doesn't matter.

Once the client has opened the queue, it enters a program loop where it will send messages to the server. It begins by allocating a shared memory segment. Then it uses `DosGiveSeg` and the `MAKEP` macros to create a pointer to the shared memory that the queue owner—the server process—can use. The client calls `DosWriteQueue` to place the pointer, the length of the message, and the element priority in the queue. The client immediately frees the shared memory segment, which won't be discarded until the server frees it. Then the client prints a message to show that it successfully placed a message in the queue to the server. If the oper-

ator has pressed a key, it closes the queue and exits the program. Otherwise, it increments the priority (or resets it to 0) and sleeps for a while before continuing at the top of the program loop.

Although you can run only one instance of the server program, you can run a client program in every other session. If you are willing to forgo the acknowledgment from a client process that it is sending messages, you can run DETACHED instances of clients that can also communicate with the server. In fact, you can DETACH the server, which will invisibly accept messages from several clients.

One note about priorities: try pressing Control-S in the server's session while the client processes are generating messages. Then, after a few moments, press another key to get the server restarted. The server will continue with the highest-priority elements first, gradually working down to lower-priority elements until it catches up.

Signals

OS/2 signals are IPC mechanisms that notify a process that an external event has occurred. Signals are routinely used by OS/2 to communicate with a process. For instance, if a user presses Control-C or Control-Break to terminate a process, OS/2 will generate the signal SIG_CTRLBREAK or the signal SIG_CTRLC. If the process has installed a signal handler, it can field these signals in any manner it wishes. Otherwise, the command processor parent of the process will receive the signal, translate it into the SIGKILLPROCESS termination signal, and pass it to the process. OS/2 may also terminate a process by sending SIGKILLPROCESS to it directly (if you select Close in

Figure 3: Source Code for Shared Memory

SHMEM.MAK

```
# make file for shmem.c
#
COPT=/Lp /W3 /Zp16 /G2s /I$(INCLUDE) /Alf
shmem.exe: shmem.c shmem.mak
cl $(COPT) shmem.c /link /co /noi libbcat
markexe windowcompat shmem.exe
```

SHMEM.C

```
/* shmem.c RBS 5/2/89
This program demonstrates the use of shared memory. To use it, run
more than one copy of the program in different sessions with:

SHMEM
*/
#define INCL_DOS
#define INCL_KBD
#define INCL_ERRORS

#include<os2.h>
#include<stdio.h>
#include<memory.h>

#define SHAREDMEM "\\sharemem\\shmem.mem"
#define SEMNAME "\\sem\\ssm.sem"
#define SHAREDMEMSIZE 200

void error_exit(USHORT err, char *msg);
PID Os2GetPid(void);
void main(void);

void main(void)
{
    HSYSSEM semhandle;
    SEL selector;
    USHORT retval;
    int creator = FALSE;
    KBDKEYINFO kbdkeyinfo;
    PCH p;
    PID pid;

    pid = Os2GetPid();

    /* first process will successfully execute this code and create the
    semaphore */
    if(retval = DosCreateSem(CSEM_PUBLIC, &semhandle, SEMNAME))
    {
        if(retval != ERROR_ALREADY_EXISTS)
            error_exit(retval, "DosCreateSem");

        /* second process will open the semaphore previously created */
        else if(retval = DosOpenSem(&semhandle, SEMNAME))
            error_exit(retval, "DosOpenSem");

        if(retval = DosGetShrSeg(SHAREDMEM, &selector))
            error_exit(retval, "DosGetShrSeg");

        DosSemWait(semhandle, SEM_INDEFINITE_WAIT);
        DosSemRequest(semhandle, SEM_INDEFINITE_WAIT);
    }
    else
    {
        /* then first process will wait here until second process executes
        the code in the loop below */
        if(retval = DosAllocShrSeg(SHAREDMEMSIZE,
        SHAREDMEM, &selector))
            error_exit(retval, "DosAllocShrSeg");

        DosSemSet(semhandle);
        creator = TRUE;
    }

    p = MAKEP(selector, 0); /* create pointer to shmem */
}
```

Figure 3

```

if (!creator)
    memset(p, '\0', SHAREDMEMSIZE);

while(TRUE)
/* both processes will continue to execute in this loop*/
{
    if (*p)
        printf("%s (%u) received: %s\n", \
            (creator ? "Owner" : "User"), pid, p);

    sprintf(p, (creator ?
        "from %u, the owner of this shared \
        memory...Hello!" : "from %u, \
        a user of this shared memory...Hello!"), pid);

    DosSemClear(semhandle);

    KbdCharIn(&kbdkeyinfo, IO_NOWAIT, 0);
    /* check for key press */
    if (kbdkeyinfo.fbStatus & FINAL_CHAR_IN)
    /* if pressed, break out */
        break;

    DosSleep(150L);
    DosSemRequest(semhandle, SEM_INDEFINITE_WAIT);
}

DosCloseSem(semhandle); /* close the semaphore */
DosFreeSeg(selector);
DosExit(EXIT_PROCESS, 0); /* and exit */
}

PID Os2GetPid(void) /* returns process id */
{
    PIDINFO pidinfo;

    DosGetPID(&pidinfo); /* get process id */
    return pidinfo.pid; /* return it */
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

```

the System Icon pulldown menu on a Presentation Manager window, the window's command processor will send SIGKILLPROCESS to the application). OS/2 also sends a SIGKILLPROCESS signal to a child process when the parent process uses DosKillProcess to terminate the child.

Signals can also be used in OS/2 for simple communications between two applications. OS/2 provides three signal flags, SIG_PFLG_A, SIG_PFLG_B, and SIG_PFLG_C, that an application can use to signal another.

Signal handling under OS/2 is similar to interrupt handling under DOS². Under DOS, an external event generates an

interrupt to the CPU. The CPU then executes a predefined interrupt handling routine, which DOS or the machine BIOS provides. A DOS application can substitute its own interrupt handler to process interrupts while the application is running. Then it will (we hope) reset the predefined interrupt handler when it is through or about to terminate.

Under OS/2, external events generate signals. OS/2 provides a predefined signal handler for each type of signal, but an application can substitute or register its own signal handler instead. Then the application can reset the original signal handler when it is done. However, since a signal handler is specific to an

application, it doesn't matter if the application doesn't replace the signal handler before terminating. An application can use DosSetSigHandler to set up its own signal-handling routine by specifying the address of the signal handler function and the type of signal to be trapped.

An application must register a signal handler with OS/2 for each type of signal. If the process has not registered a signal handler by the time the signal is generated, OS/2 will pass the signal handler to the default signal handler—usually the command processor for the session, either CMD.EXE or the Presentation Manager shell (depending how the application was started).

When OS/2 passes a signal to an application that has registered its own signal handler for that signal, it switches the application's main thread to the code of the signal-handling routine. Since signals are time-critical events that require an immediate response, OS/2 will interrupt the main thread even if it is executing a critical section of code (that is, one defined by a call to DosEnterCritSec and a subsequent call to DosExitCritSec).

If the main thread is waiting for the return of an API function when it is switched to the signal handler, OS/2 will force the thread to abort the call rather than wait for the function to complete. Once the main thread has processed the signal, OS/2 will switch it back to its own code. If an API function is aborted, the function call will return an error code. The exceptions are API functions that perform file I/O: OS/2 places calls to these functions in a blocked state when they are called. Thus OS/2 can switch the calling thread back to the middle of an API file I/O function without aborting the function call.

The lesson to be learned from

these applications is that if you are writing an application that uses a signal handler, do not use the main thread for any work that would be corrupted if a signal interrupts it. To be safe, have the main thread start one thread of execution that executes the remaining code and starts any other threads. Then the main thread should install the signal handler and indefinitely block on a call to DosSleep. You can use DosHoldSignal to have the main thread temporarily ignore a signal while it is performing some uninterruptible processing, but you should not do this as a general practice.

An application should never install a signal handler that will ignore SIG_CTRL C, SIG_CTRL BREAK, or SIGKILLPROCESS, even if it ignores the signal flags. For instance, if the signal handler ignores SIGKILLPROCESS, you will not be able to terminate it without rebooting your machine. Upon receiving any of these three signals, the process should free its resources, notify any child processes to terminate, and terminate itself. You can use DosExitList to ensure that this occurs properly.

A well-written OS/2 application can, however, use the three signal flags for communicating with other processes. These are not flags that communicate a state (as semaphores do), nor are they flags that a process can manipulate. Instead, these flags signal a process that something has happened and allow it to act accordingly.

If you are writing an application that will handle signals, you can install a signal handler with DosSetSigHandler. This function can specify five different ways of reacting to a specific signal. The process can:

- replace the current signal handler with another
- acknowledge the signal

Figure 4: Source Code for a Queue Server

```

Q.H
/* q.h RRS 5/1/89
   queue server/client common header file
*/

#define QUEUENAME        "\\queues\\qserver.que"

OSERVER.MAK
# make file for qserver.c
#
COPT=/Lp /W3 /xpiel /G2e /IS (INCLUDE) /Alfw

qserver.exe: qserver.c qserver.mak q.h
cl $(COPT) qserver.c /link /co /noi llibcmt
markexe windowcompat qserver.exe

OSERVER.C
/* qserver.c RRS 5/1/89
   This program demonstrates the use of queues and creates a server
   process that client processes can pass messages to. To use it, start
   one instance of this program with:

   QSERVER

   Then start multiple instances of QCLIENT.C with:

   QCLIENT
*/

#define INCL_DOS
#define INCL_DOSQUEUES
#define INCL_KBD
#define INCL_ERRORS

#include<os2.h>
#include<stdio.h>

#include"q.h"

void error_exit(USHORT err, char *msg);
void main(void);

void main(void)
{
    USHORT retval, qcount = 0;
    KBDKEYINFO kbdkeyinfo;
    PCB p;
    HQUEUE qhandle;
    QUEUERESULT qresult;
    USHORT ellength;
    BYTE priority;

    if(retval = DosCreateQueue(&qhandle, QUE_PRIORITY, QUEUENAME))
        error_exit(retval, "DosCreateQueue");

    printf("Server has opened queue, awaiting messages...\n");

    while(TRUE)
    {
        if(retval = DosReadQueue(qhandle, &qresult, &ellength,
            (PVOID FAR *) &p, 0x0000, DCWN_WAIT, &priority, 0L))
        {
            if(retval != ERROR_QUEUE_EMPTY)
                /* if error was not from empty queue */
                error_exit(retval, "DosReadQueue");
            continue;
        }
        DosQueryQueue(qhandle, &qcount);
        printf(
            "Server: (%u pending) message received from %u,\n",
            priority & 02u, &u\n", qcount,
            qresult.pidProcess, priority, p);
        DosFreeSeg(SELECTOROF(p));
    }
}

```

Figure 4

```

        KbdCharIn(&kbdkeyinfo, IO_NOWAIT, 0);
        /* check for key press */
        if (kbdkeyinfo.fbStatus & FINAL_CHAR_IN)
            /* if pressed, break out */
            break;
    }
    DosCloseQueue(qhandle);
    DosExit(EXIT_PROCESS, 0); /* and exit */
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

```

QCLIENT.MAK

```

# make file for qclient.c
#

COPT=/Lp /W3 /Zpie1 /G2s /I$(INCLUDE) /Alfw

qclient.exe: qclient.c qclient.mak q.h
cl $(COPT) qclient.c /link /co /noi llibcm
markexe windowcompat qclient.exe

```

QCLIENT.C

```

/* qclient.c RRS 5/1/89
This program demonstrates the use of queues. To use it, you should first
start QSERVER.C with:

QSERVER

Then you can run multiple copies of this program with:

QCLIENT
*/

#define INCL_DOS
#define INCL_KBD
#define INCL_ERRORS

#include<os2.h>
#include<string.h>
#include<stdio.h>
#include"q.h"

char *p = "Hello!";
#define MAX_PRIORITY    15

void error_exit(USHORT err, char *msg);
PID Os2GetPid(void);
void main(void);

void main(void)
{
    USHORT retval;
    KBDKEYINFO kbdkeyinfo;
    HQUEUE qhandle;
    BYTE priority = 0;
    SEL selector, qowner, sel;
    PID qowner;
    PCB ptr;
    int len;
    PID pid;

    pid = Os2GetPid();

    while(TRUE)
    {
        if (! (retval = DosOpenQueue(&qowner, &qhandle, QUEUE_NAME)))
            break;
        if (retval != ERROR_QUEUE_NAME_NOT_EXIST)
            error_exit(retval, "DosOpenQueue");
    }
}

```

- return an error to the sender (indicating that the receiver has refused to accept the signal)
- ignore the signal
- remove the signal handler

To call `DosSetSigHandler`, your program must specify the signal, the action to take, and the address of the signal handler function being installed. If you are supplying the latter, `DosSetSigHandler` will return the address of the current signal handler and its action, so that your application can restore these later.

The signaling process should use `DosFlagProcess` to signal the receiver process, but to use this function it must know the process ID of the receiver. In addition, `DosFlagProcess` allows the sender to send a 2-byte unsigned value to the receiver. The content of this argument is left to the discretion of the application; this is not unlike the 2-byte value that a queue sender can pass to a queue owner.

Once the signal has been generated, OS/2 will preempt the receiver's primary thread and direct it to execute the code of the new signal handler. The signal handler should immediately issue a call to `DosSetSignalHandler` to acknowledge the signal. This action resets the signal handler and allows the process to accept the signal again (the receiver can't receive that signal again until it has cleared the signal handler). If the signal handler does not acknowledge the signal, `DosFlagProcess` will return an error code to the sender indicating that the receiver refused to accept the signal. Once the receiver process no longer needs to handle the new signal, `DosSetSignalHandler` should be used to restore the previous signal handler.

The signal handler routine always has the same form:

```
void APIENTRY sighandler
(USHORT arg, USHORT num);
```

(This line was broken due to space limitations, but should run on one line—Ed.) You can use any name you wish for the function and arguments, but you must declare the function this way. OS/2 will supply the contents of the two arguments. You never actually call the signal handler function in your code—when your program receives the appropriate signal, OS/2 will set up the stack as if the function had been called and jump to the first line of code inside the function. The first argument specifies the type of signal that was sent. If it is a flag signal, the second argument will be the discretionary value supplied to DosFlagSignal by the sender. Otherwise, OS/2 will set this argument to 0.

When the receiver's signal handler is invoked, OS/2 will load the CS, IP, SS, SP, and flag registers with new values to represent the code and stack for the signal-handling routine. The signal handler should terminate with a far return so that OS/2 can resume execution of the interrupted thread. Fortunately, if you are writing your application in C, this process is transparent to the application and your code, since the function was declared with the APIENTRY type modifier. OS/2 will execute the far return for you when it reaches the end of the function.

What happens if your program receives a signal while the signal handler is active? If the receiver's signal handler hasn't acknowledged or cleared the signal yet, DosFlagProcess will return an error to the signal-sending process. If the receiver's signal handler has acknowledged the signal but is still active (that is, execution has not returned to the primary

thread), the OS/2 error handler will trap the second occurrence of the signal. In other words, the receiver will not know that the signal was sent, and DosFlagProcess will not return an error to the sender (not yet, at least). However, if the sender generates the signal a third time while the receiver's error handler is still active after acknowledging the first signal, DosFlagProcess will return

Figure 4

```

else
{
    DosSleep(1000L);
    continue;
}

printf("Client (%u) has opened queue, now messaging...\n", pid);

while(TRUE)
{
    if (retval = DosAllocSeg(len = (strlen(p)+1), &selector,
        SEG_GIVEABLE))
        error_exit(retval, "DosAllocSeg");

    ptr = MAKEP(selector, 0);
    strcpy(ptr, p);
    if (retval = DosGiveSeg(selector, qowner, &qownersel))
        error_exit(retval, "DosGiveSeg");

    ptr = MAKEP(qownersel, 0);

    if (retval = DosWriteQueue(qhandle, 0, len,
        (PBYTE)ptr, priority))
        error_exit(retval, "DosWriteQueue");

    DosFreeSeg(selector);
    printf("Client (%u) sent message, priority %u\n",
        pid, priority);

    KbdCharIn(&kbdkeyinfo, IO_NOWAIT, 0);
    /* check for key press */
    if (kbdkeyinfo.fbStatus & FINAL_CHAR_IN)
        /* if pressed, break out */
        break;
    if (++priority > MAX_PRIORITY)
        priority = 0;
    DosSleep(50L);
}

DosCloseQueue(qhandle);
DosExit(EXIT_PROCESS, 0); /* and exit */
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

PID Os2GetPid(void) /* returns process id */
{
    PIDINFO pidinfo;

    DosGetPID(&pidinfo);
    return pidinfo.pid; /* return it */
}

```

THE SIGNALING PROCESS SHOULD USE DOSFLAGPROCESS TO SIGNAL THE RECEIVER PROCESS, BUT TO USE THIS FUNCTION IT MUST KNOW THE PROCESS ID OF THE RECEIVER.

Figure 5: Source Code for Signal Handler

SIG.MAK

```

! make file for sig.c
!
COPT=/Lp /W3 /Zpiel /G2s /IS(INCLUDE) /Alfw

sig.exe: sig.c sig
cl $(COPT) sig.c /link /co /noi llibcmt
markexe windowcompat sig.exe

```

SIG.C

```

/* sig.c RBS 5/1/89 signal handler example
This program lets you observe how one process can use the OS2
signaling mechanism to pass signals between two processes.
To use it, first run the program in one session with:

    SIG

Note the process ID it prints. This will be the receiver of
the signals. Then you can run subsequent instances of SIG in
other sessions by passing them the process ID of the first
instance as a command line argument:

    SIG n

where 'n' is the process ID printed by the first instance. */

#define INCL_DOS
#define INCL_VIO
#define INCL_KBD
#define INCL_ERRORS

#include<os2.h>
#include<nt\stdio.h>
#include<nt\memory.h>
#include<nt\string.h>
#include<nt\stdlib.h>
#include<nt\process.h>

#define SEMNAME "\\sem\\ssm.sem"
#define THREADSTACK 400
char keyboard_thread_stack[THREADSTACK];

unsigned pid = 0;
unsigned signaler = FALSE;
USHORT prevaction0, prevaction1, prevaction2, prevaction3, prevaction4,
prevaction5;
PFNSIGHANDLER prevhandler0, prevhandler1, prevhandler2, prevhandler3,
prevhandler4, prevhandler5;

#define MAXSIGS 3

char *signames[MAXSIGS] = {"Flag A", "Flag B", "Flag C"};
USHORT signal[MAXSIGS] = {PFLG_A, PFLG_B, PFLG_C};

void main(int argc, char **argv);
PID OS2GetPid(void);
void APIENTRY sighandler(USHORT arg, USHORT num);
void error_exit(USHORT err, char *msg);
void keyboard_thread(void);

void APIENTRY sighandler(USHORT arg, USHORT num)
{
    USHORT retval;
    PFNSIGHANDLER prevhandler = NULL;
    USHORT prevaction = 0;

    /* acknowledge immediately */
    retval = DosSetSigHandler((PFNSIGHANDLER)sighandler,
        &prevhandler, &prevaction, SIGA_ACKNOWLEDGE, num);

    switch(num)
    {

```

ERROR_SIGNAL_PENDING to the receiver. Consequently, OS/2 can only handle two signals per process at a time.

Sample Signal Program

The sample SIG.C program (see Figure 5) shows you how to create a process that can signal another process. In keeping with the sample semaphore and shared memory programs, I have used one body of source code to create a program that you can run simultaneously in different sessions.

The first instance of SIG will always be the receiver. When you run it, SIG will display its process ID and attempt to create a system semaphore. If successful, it will start a keyboard thread. The keyboard thread blocks on a call to KbdCharIn and terminates the process immediately upon its return. The user can then terminate the receiver instance of SIG at any time by pressing a key. Since the KbdCharIn call is in a separate thread, the main thread can install the signal handler and then sleep indefinitely. This eliminates the possibility of the keyboard call being interrupted by a jump to the signal handler code.

After the main thread has started the keyboard thread, it installs the signal handler. The signal-handling routine will process all signals except the one used to detect a broken pipe (DosSetSigHandler will return an error if you attempt to install a signal handler for SIG_BROKENPIPE without opening the pipe first). Once it has installed the signal handler, the main thread enters a loop in which it sleeps for 10 seconds, wakes up, and sleeps again.

To generate some signals, you must run at least one other instance of the program in a different session. When you run a signaling instance of SIG, you must pass it the process ID of the

receiver instance as a command line parameter. Therefore, if the receiver prints a process ID of 50 when you start it, run each subsequent instance with:

SIG 50

After printing its own process ID, each signaler attempts to create the semaphore. When the signaler fails (since the receiver has already created the semaphore) it begins to execute the sender code in the program. It reads the receiver's process ID from the command line and enters the program loop, where it calls `DosFlagProcess` repeatedly to signal the receiver, switching to a different signaling flag each time, and prints a message to the screen with each pass. As it goes through the loop, it checks for any keyboard input and terminates if the user has pressed a key. Then it sleeps briefly before returning to the top of the loop.

The receiver process remains idle until it receives a signal. When it receives a signal, OS/2 switches the main thread to the code of the signal handler function, `sighandler`. This function immediately acknowledges the signal with a call to `DosSetSigHandler`, prints a message identifying the signal, and prints a subsequent message showing whether it successfully acknowledged the signal. Then the function returns and OS/2 switches the thread back to its own code.

Note that you can't terminate the receiver with a Control-C or Control-Break. The receiver will acknowledge these signals but refuse to terminate. You can, however, end the program by selecting Close from the OS/2 Presentation Manager system icon or by selecting Task/Close from the Presentation Manager Task Manager window. If you do this, the program will print a message acknowledging the sig-

Figure 5

```

case SIG_CTRLC:
    printf("tu received ^C...", pid);
    break;
case SIG_KILLPROCESS:
    printf("tu notified of pending termination,
    terminating...", pid);
    DosExit(EXIT_PROCESS, 0);
    break;
case SIG_CTRLBREAK:
    printf("tu received ^Break...", pid);
    break;
case SIG_PFLG_A:
    /* FLAG_A received */
    printf("tu received signal, Flag A,
    arg=tu...", pid, arg);
    break;
case SIG_PFLG_B:
    printf("tu received signal, Flag B,
    arg=tu...", pid, arg);
    break;
case SIG_PFLG_C:
    printf("tu received signal, Flag C,
    arg=tu...", pid, arg);
    break;
default:
    printf("tu received unknown signal (%u),
    arg=tu", pid, num, arg);
    break;
}

if(retval)
    printf("unable to acknowledge signal,
    retval=tu.\n", retval);
else
    printf("acknowledged.\n");
return;
}

void main(int argc, char **argv)
{
    unsigned retval, receiver_pid, signal = 0;
    HSYSSEM semhandle;
    KBDKEYINFO kbdkeyinfo;

    pid = OS2GetPid();
    printf("%s loaded, pid=tu\n", argv[0], pid);

    if(retval = DosCreateSem(CSEM_PUBLIC, &semhandle, SEMNAME))
    {
        if(retval != ERROR_ALREADY_EXISTS)
            error_exit(retval, "DosCreateSem");

        signaler = TRUE; /* set flag */
        if(argc != 2)
            error_exit(0, "main");
        receiver_pid = atoi(argv[1]);
    }
    else
    {
        if(_beginthread(keyboard_thread, keyboard_thread_stack,
        THREADSTACK, NULL) == -1)
            error_exit(-1, "_beginthread");

        /* set up signal handler to be sighandler function, pass address of
        prevhandler and prevaction even though there weren't any previous. */
        if(retval = DosSetSigHandler(sighandler, &prevhandler0,
        &prevaction0, SIGA_ACCEPT, SIG_CTRLC))
            error_exit(retval, "DosSetSigHandler(0)");
        if(retval = DosSetSigHandler(sighandler, &prevhandler1,
        &prevaction1, SIGA_ACCEPT, SIG_KILLPROCESS))
            error_exit(retval, "DosSetSigHandler(1)");
        if(retval = DosSetSigHandler(sighandler, &prevhandler2,
        &prevaction2, SIGA_ACCEPT, SIG_CTRLBREAK))
            error_exit(retval, "DosSetSigHandler(2)");
        if(retval = DosSetSigHandler(sighandler, &prevhandler3,
        &prevaction3, SIGA_ACCEPT, SIG_PFLG_A))
            error_exit(retval, "DosSetSigHandler(3)");
    }
}

```

Figure 5

```

        if (retval = DosSetSigHandler(sighandler, &prevhandler4,
        &prevaction4, SIGA_ACCEPT, SIG_PFLG_B))
            error_exit(retval, "DosSetSigHandler(4)");
        if (retval = DosSetSigHandler(sighandler, &prevhandler5,
        &prevaction5, SIGA_ACCEPT, SIG_PFLG_C))
            error_exit(retval, "DosSetSigHandler(5)");
    }

    while (TRUE)
    {
        if (signaler) /* if another copy is out there */
        {
            /* signal it (not its children) with FLAG_A */
            if (retval = DosFlagProcess(receiver_pid,
            FLGP_PID, signal[signal], pid))
                error_exit(retval, "DosFlagProcess");
            else
                printf("Sender (%u): sent signal to %u: %s\n",
                pid receiver_pid, signames[signal]);
            if (++signal >= MAXSIGS)
                signal = 0;

            /* check for key press */
            retval = KbdCharIn(&kbdkeyinfo, IO_NOWAIT, 0);
            if (!retval || (retval != ERROR_SIGNAL_PENDING))

                /* if pressed, break out */
                if (kbdkeyinfo.fbStatus & FINAL_CHAR_IN)
                    break;
            DosSleep(100L); /* sleep again */
        }
        else
            DosSleep(10000L); /* don't wake too often */
    }
}

PID OS2GetPid(void)
{
    PLINFOSEG ldt;
    SEL gdt_descriptor, ldt_descriptor; /* infoseg descriptors */

    if (DosGetInfoSeg(&gdt_descriptor, &ldt_descriptor))
        return 0;

    ldt = MAKEPLINFOSEG(ldt_descriptor);
    return ldt->pidCurrent;
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

void keyboard_thread(void)
{
    KBDKEYINFO kbdkeyinfo;

    while (TRUE)
    {
        if (!KbdCharIn(&kbdkeyinfo, IO_WAIT, 0))
            DosExit(EXIT_PROCESS, 0);
    }
}

```

nal before terminating. You can also end the program by pressing a key when the receiver's session is in the foreground. You might want to experiment with running several instances of SIG, each signaling the first instance. This experiment should

give you a good idea of how signaling works and how to use it.

Finally, I should mention that although signaling is fast, the system semaphore is the preferred means of setting a flag that can be used by two different processes. If speed is critical and

the other caveats associated with signals are not a problem, by all means go ahead and use them. But for most signaling, system semaphores are a more elegant solution.

Device Monitors

Character device monitors are not a form of OS/2 IPC. They do, however, allow an application to monitor and manipulate the data sent to and from a character device. In this way you can create a process that can monitor or modify the data stream between a device driver and its associated subsystem, whether or not the monitor process is in the foreground session. While monitors are not a part of OS/2 IPC, they do allow one application to control and manipulate the data that another application receives from a device.

The prototypical example of a device monitor is a keyboard monitor. You can use an OS/2 keyboard monitor to create a program that activates when the user presses a hot key, just as with many of the popular DOS terminate-and-stay-resident programs. I'll limit the discussion of monitors here to what you need to know to create programs that do just that.

A device monitor only monitors a device's interaction with the screen group in which you install it. You can, however, install a monitor in more than one screen group at a time. If you install a keyboard monitor in one of the Presentation Manager's sessions, it will monitor the keyboard device stream for the entire Presentation Manager screen group. You can use monitors on any character device, including the keyboard, the mouse, and the printer, but you cannot monitor a block device (like a disk drive) or the asynchronous communications device driver. Usually, you can control only the data stream

flowing between the device driver and the API subsystem for that driver. However, a printer monitor will let you control the data flowing in either direction.

When you install a device monitor, OS/2 places it in the data stream between the device driver and the buffers used by the associated API subsystem. A keyboard monitor, for example, will receive data from the KBD\$ device driver and pass data to the keyboard (KBD) subsystem. If you install more than one monitor, OS/2 will pass the data in the stream from one monitor to the next. This collection of monitors is known as the monitor chain. Together a set of monitors can filter the data placed in a session's API buffer for the device being monitored.

There are several steps that a program must take to create and install a character device monitor. The monitor process should issue a call to `DosMonOpen` first, which will return a device handle for the data stream to be monitored. Then the program should call `DosMonReg`, which will register the process as a device monitor for the data stream. A call to `DosMonReg` causes the device driver to create a monitor dispatcher. The monitor dispatcher will divert the data stream from the device driver to the monitor and from the monitor to the API buffer (or the next monitor in the chain); `DosMonReg` will also register the monitor's input and output buffers with the monitor dispatcher.

Once you've registered the monitor, the device driver's monitor dispatcher will automatically place records from the data stream into the monitor's input buffers. The monitor examines the data in its buffers, and can either modify it or leave it alone. Then it must efficiently transfer the data from its input

buffer to its output buffer by means of calls to `DosMonRead` and calls to `DosMonWrite`. The monitor dispatcher will automatically transfer the data to the API buffers or the next monitor in the chain. Note that a monitor can optionally consume a data record if it does not want it passed on to the application. When that happens, the application never receives the data.

Because a monitor can affect the device information received by applications, you should take care in designing and writing one. You can use `DosMonReg` to set the monitor's position in the monitor chain (first, last, or next available). OS/2 will always place the first monitor to register for the first position at the front of the monitor chain. If another monitor subsequently registers for first position, OS/2 will place it after any other monitor that has already registered for that position. Likewise, OS/2 will place the first monitor that registers for last position last, and subsequent monitors vying for that position will be next to last, and so on. If a monitor registers for the next available position, it will be placed after any monitors that requested first position and before any that requested last position. You can usually place a keystroke monitor almost anywhere in the monitor chain so long as other monitors do not remove or modify the data for which it's looking.

A monitor should be efficient; it should read from, modify, and write to the data stream as quickly as possible. In addition, a monitor should never take any action that might disrupt the data stream. As a rule, you should dedicate one thread of the monitor process to read to and write from the data stream and another thread to react to the data content.

`DosMonOpen` should be called only once in a program.

Therefore if your application is going to use multiple monitors, place the monitor code from the call to `DosMonReg` through the `DosMonRead/DosMonWrite` loop in a new thread. The portion of the thread that performs the `DosMonRead` and `DosMonWrite` calls should not have to wait for I/O services or perform operations that may block it. If the monitor thread is blocked, the monitor will stop the entire data stream, halting the flow of data from the device driver to every process in the screen group until the monitor thread becomes unblocked. If the monitor hangs, it can hang the entire screen group.

A monitor thread's priority should be set to help ensure that the OS/2 task scheduler dispatches the monitor thread before it dispatches any threads accessing the API buffer. The monitor should run at the lower levels of the time-critical (the highest) priority category. If the monitor's priority is too low, it will waste CPU cycles and not deliver data from the device driver fast enough. Then the threads reading the API buffer will block longer than necessary, since they will be waiting for the arrival of the data in the buffer.

Note that I said to place the monitor at the lower level of the time-critical priority category. Threads that service the hardware device drivers usually use the top of the time-critical category. The monitor must wait for these threads to deliver data from the device to the monitor chain. If you set the monitor at the same level, it will use too

**A DEVICE MONITOR
ONLY MONITORS A
DEVICE'S
INTERACTION WITH
THE SCREEN GROUP
IN WHICH YOU
INSTALL IT. YOU CAN,
HOWEVER, INSTALL A
MONITOR IN MORE
THAN ONE SCREEN
GROUP AT A TIME.**

Figure 6: Source Code for Keyboard Device Monitors

MONITOR.MAK

```
# make file for monitor.c
#
COPT=/Lp /W3 /Zp16 /G2s /I$(INCLUDE) /Alf
monitor.exe: monitor.c monitor.mak
cl $(COPT) monitor.c /link /oo /noi llibcm
markexe windowcompat monitor.exe
```

MONITOR.C

```
/* monitor.c RMS 5/1/89
This program demonstrates the use of character device monitors on the
keyboard. It creates a keyboard monitor that you can run from any
session or the DETACHED session. Once you run the monitor with:

    MONITOR

you can activate the monitor with Alt-F10. This will create a pop-up
screen which will wait for any key press to end.

You can terminate the monitor with Ctrl-F10. */

#define INCL_DOS
#define INCL_VIO
#define INCL_KBD
#define INCL_ERRORS

#include <os2.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

unsigned long semhandle = 0L;
unsigned char HotKey = 0x71; /* HotKey = Alt-F10 */
unsigned char TerminateKey = 0x67; /* TerminateKey = Ctrl-F10 */
unsigned program_active = TRUE;
unsigned popup_active = FALSE;

MONIN monInbuf;
MONOUT monOutbuf;

#define THREADSTACK 400
char monitor_stack[THREADSTACK];

#define MONINBUFSIZE (sizeof(monInbuf)-sizeof(monInbuf.cb))
#define MONOUTBUFSIZE (sizeof(monOutbuf)-sizeof(monOutbuf.cb))
#define RELEASE 0x40 /* bit mask to distinguish */
#define MAXMILLISECONDS 20000L
#define SLEEPTIME 1000L

typedef struct _keypacket /* KBD monitor data record */
{
    unsigned mflags;
    KBDKEYINFO cp;
    unsigned ddflags;
} KEYBUFF;

void error_exit(USHORT err, char *msg);
void main(void);
void keyboard_monitor(void);

void main(void)
{
    USHORT popupwait = (VP_WAIT | VP_OPAQUE);
    KBDKEYINFO kbdkeyinfo;
    long milliseconds;

    DosSemSet(&semhandle);

    if( _beginthread(keyboard_monitor, monitor_stack, THREADSTACK,
        NULL) == -1)
        error_exit(-1, "_beginthread");
    while(TRUE)
    {
```

many CPU cycles, since the dispatcher will schedule it on an equal basis with the device driver threads. Furthermore, if the time-critical priority category is overloaded with monitors, the device drivers will not be able to meet the demands of the devices.

Keyboard Monitor

The sample MONITOR.C program (see Figure 6) shows how you can install a simple keyboard monitor. This monitor examines the data stream from the keyboard device driver and displays a pop-up screen whenever the user presses the Alt-F10 hot key. The pop-up screen waits for another key press, then ends the pop up. If the user presses the Ctrl-F10 termination key, the monitor removes itself from the keyboard monitor chain and terminates itself.

MONITOR consists of two threads. The main thread begins by setting a RAM semaphore and launching the monitor thread. The monitor thread uses the RAM semaphore to signal the main thread to create the pop up. After the main thread starts the monitor thread, it enters a program loop and blocks on a call to DosSemRequest. When the monitor thread clears the semaphore, the main thread will become active. It will call VioPopUp to create a pop-up screen, print a message, and enter another program loop.

In the next program loop, the main thread repeatedly checks for a key press from the operator and breaks out of the loop if a key is available. If not, the thread sleeps for one second. Each time it goes through the loop, the thread increments a counter; when 20 seconds have passed (or rather, twenty 1-second sleep intervals), the thread will break out of the loop. Once out of the loop, the thread will end the pop-up screen and block

on the RAM semaphore again. Using the inner loop allows this thread to wait until a user presses a key without waiting indefinitely while a pop-up screen is active. Although OS/2 will turn over the pop-up session to a pending VioPopUp call after 30 seconds, it makes no sense for the thread to own this resource for more than 20 seconds.

If the monitor thread clears the program_active variable and the RAM semaphore, the main thread will break out of the main program loop and then terminate the program.

The monitor thread itself is a fairly bland example of a keyboard monitor. First, it calls DosMonOpen to get a device handle for the keyboard device driver. Then the thread calls DosGetInfoSeg to get the program's screen group number, which is needed by DosMonReg. It registers its input and output buffers with DosMonReg and raises the thread's priority with DosSetPrtty. Then the thread enters the main monitor loop, reading keyboard data from the KBD device driver with DosMonRead and writing the data to the KBD API buffer with DosMonWrite (assuming no other monitors are installed).

Between each read and write, the monitor thread checks to see if the scan key for a keystroke is the hot key required to generate the pop-up screen in the main thread. If so, the monitor thread sets a variable, popup_active, clears the RAM semaphore, and returns to the top of the loop.

By clearing the RAM semaphore, the monitor thread activates the main thread and uses the popup_active variable to determine whether the pop-up screen is active. If it is active, the monitor thread will pass the key to the API buffers, even if it is the hot key.

If the monitor thread discovers the termination key in the

Figure 6

```

DosSemRequest(&semhandle, SEM_INDEFINITE_WAIT);
/* wait for semaphore */

if(!program_active) /* time to terminate? */
    break; /* then get out */
VioPopUp(&popupwait, 0);
printf("You made it...press a key");

/* wait 20 seconds for keystroke, then break out */
for( milliseconds = 0L; milliseconds <
    MAXMILLISECONDS; milliseconds += SLEEPTIME)
{
    KbdCharIn(&kbdkeyinfo, IO_WAIT, 0);
    /* check for key press */
    if(kbdkeyinfo.fbStatus & FINAL_CHAR_IN)
        /* if pressed, break out */
        break;
    DosSleep(SLEEPTIME); /* sleep for a while */
}

VioEndPopUp(0); /* and the popup */
popup_active = FALSE;
}
DosExit(EXIT_PROCESS, 0); /* kill the process */
}

void keyboard_monitor(void)
{
    USHORT retval;
    SEL gdt_descriptor, ldt_descriptor; /* infoseg descriptors */
    KEYBUFF keybuff;
    unsigned count;
    HKBD KBDHandle;
    PCINFOSEG gdt;
    PLINFOSEG ldt;

    monInbuf.cb = MONINBUFSIZE;
    monOutbuf.cb = MONINBUFSIZE;

    /* obtain a handle for registering buffers */
    if(retval = DosMonOpen("KBD", &KBDHandle))
        error_exit(retval, "DosMonOpen");

    if(DosGetInfoSeg(&gdt_descriptor, &ldt_descriptor))
        error_exit(retval, "DosGetInfoSeg");

    gdt = MAKEPGINFOSEG(gdt_descriptor);
    ldt = MAKEPLINFOSEG(ldt_descriptor);

    /* register the buffers to be used for monitoring */
    if(retval = DosMonReg(KBDHandle, (PBYTE)&monInbuf,
        (PBYTE)&monOutbuf, MONITOR_BEGIN, gdt->sgCurrent))
        error_exit(retval, "DosMonReg");

    /* raise thread priority */
    DosSetPrtty(PRTYS_THREAD, PRTYC_TIMECRITICAL, 0, 0);

    for(keybuff.cp.chChar = 0; ; ) /* loop thru all keyboard in */
    {
        count = sizeof(keybuff);

        /* read keyboard */
        if(retval = DosMonRead((PBYTE)&monInbuf, IO_WAIT,
            (PBYTE)&keybuff, (PUSHORT)&count))
            error_exit(retval, "DosMonRead");

        /* if make on our key */
        if(keybuff.cp.chChar == 0)
            if(!popup_active && (keybuff.cp.chScan == \
                HotKey) && !(keybuff.ddflags & RELEASE))
            {
                popup_active = TRUE;
                DosSemClear(&semhandle);
                continue;
            }
    }
}

```

Figure 6

```

else if ((keybuff.cp.chScan == TerminateKey) \
        && !(keybuff.ddflags & RELEASE))
{
    program_active = FALSE;
    break;
}

if (retval = DosMonWrite((PBYTE)&monOutbuf,
                        (PBYTE)&keybuff, count))
    error_exit(retval, "DosMonWrite");
}

DosMonClose(KBDHandle);          /* close the monitor */
DosSemClear(&semhandle);
DosExit(EXIT_THREAD, 0);        /* kill the thread */
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

```

Figure A: Source Code for a Named Pipe Server

NPSEVER.MAK

```

# make file for npserver.c
#
%COPT=/Lp /W3 /Zp /Zl /G2s /Ox /I$(INCLUDE)      /Alfw
COPT=/Lp /W3 /Zp1 /G2s /I$(INCLUDE) /Alfw

npserver.exe: npserver.c npserver.mak
cl $(COPT) npserver.c /link /co /noi 11ibcm
markexe windowcompat npserver.exe

```

NPSEVER.C

```

/* npserver.c RMS 5/1/89
This program demonstrates the use of named pipes by creating a
server process which will create more than instance of a named pipe.
To see how this works, run this program with:

NPSEVER

Then run multiple instances of the NPCLIENT program in other
sessions. You can also run the NPMODE program in the compatibility
box to talk to the server. */

#define INCL_DOS

#include<os2.h>
#include<nt\string.h>
#include<nt\stdio.h>
#include<nt\process.h>
#include"nmpipe.h"

#define MAXSERVERS 5
#define THREADSTACKSIZE 1200

unsigned long semhandle = 0L;
typedef struct _servers
{
    char    stack[THREADSTACKSIZE];
    int     threadID;
} SERVER;

SERVER servers[MAXSERVERS];
USHORT pipeopenmode = PIPE_ACCESS_DUPLEX;
USHORT pipemode = (PIPE_WAIT | PIPE_READMODE_BYTE | PIPE_TYPE_BYTE | \
MAXSERVERS);

void main(void);
void error_exit(USHORT err, char *msg);
void server_thread(int servernum);

```

keyboard data stream, it clears the program_active variable, breaks out of the monitor loop, and closes the monitor with a call to DosMonClose. Just before the thread terminates itself, it clears the RAM semaphore. Clearing the RAM semaphore causes the main thread to unblock and shut down the process.

You can run MONITOR from any session or screen group, but you'll get the best results if you DETACH it so that it runs as a background process. You can also DETACH it from a windowed command prompt in the Presentation Manager's screen group. Doing that allows you to create a pop-up program that will appear even if you are currently using a session in the PM's screen group.

OS/2 provides enough forms of IPC—ranging from primitive signaling to sophisticated named pipes—to meet almost any programming requirement. This allows for a great deal of flexibility when developing applications that will need to share data, such as servers and communications programs. The sample programs provided with this article were designed specifically to demonstrate IPC mechanisms. The challenge is to take the code, build on it, and adapt it to fit the needs of your development project.

Named Pipes: A Welcome Latecomer

Named pipes are the newest and most powerful addition to the OS/2 IPC facilities. Originally designed to be a component of the OS/2 LAN Manager architecture, their tremendous utility and flexibility caused them to be incorporated into OS/2 Version 1.1. What distinguishes named pipes from other OS/2 IPC

facilities is their ability to transfer data across a network. An application can use them whether the process at the other end of the pipe is running on the same machine or on a machine across a network. Furthermore, named pipes are a mechanism that lets an MS-DOS operating system-based application running on a DOS workstation or in the OS/2 real-mode environment (the compatibility box) communicate with an OS/2 protected mode application (more on this below). Named pipes will probably be the most important IPC facility in OS/2.

As with anonymous pipes, a process can treat a handle to a named pipe like a file handle and use the handle with the DosOpen, DosRead, DosWrite and DosClose API functions. For the most part, however, the similarities between named pipes and anonymous pipes end there. Named pipes allow two unrelated processes to transfer data, while only related processes can use anonymous pipes. Any process that knows the name of a named pipe can open it and use it. In addition, named pipes are full-duplex, meaning the communicating processes can read from and write to them.

Named pipes are especially useful for implementing server programs, in which one process (the server) provides services for several client processes. As with queues, the process that creates the named pipe becomes the server process and owner of the pipe. Once the server has created the named pipe, any client process can open it.

Named pipes use the same naming convention as system semaphores, shared memory, and queues. In this manner, a named pipe might be called \PIPE\NAMEDP. If, however, the named pipe resides on a remote file server, the pseudo-

Figure A

```
void main(void)
{
    int i;

    DosSemSet(&semhandle);

    for(i = 0; i < MAXSERVERS; i++) /* create server threads */
    {
        if(servers[i].threadID = _beginthread(server_thread,
            servers[i].stack, THREADSTACKSIZE, (void *)i) == -1)
            error_exit(-1, "_beginthread");
        DosSleep(100L);
    }

    DosSemRequest(&semhandle, SEM_INDEFINITE_WAIT);
    /* wait for semaphore */
    DosExit(EXIT_PROCESS, 0); /* kill the process */
}

void server_thread(int servernum)
{
    char readbuf[PIPESIZE];
    HPIPE pipehandle;
    USHORT retval;
    USHORT bytesread;

    printf("Server instance %d creating Named pipe...\n", servernum);
    /* create pipe instance */

    if(retval = DosMakeNpPipe(NAMEDPIPE, &pipehandle, pipeopenmode,
        pipemode, PIPESIZE, PIPESIZE, 1000L))
        error_exit(retval, "DosMakeNpPipe");

    printf("Server instance %d waiting for Client to open pipe...\n",
        servernum);

    if(retval = DosConnectNpPipe(pipehandle))
        /* wait until connected */
        error_exit(retval, "DosConnectNpPipe");

    printf("Server instance %d: Client has opened pipe. \
        waiting for message...\n", servernum);

    while(TRUE)
    {
        /* read a message */
        if(retval = DosRead(pipehandle, readbuf, PIPESIZE, &bytesread))
            error_exit(retval, "DosRead");

        printf("Server instance %d received: \
            '%s'...acknowledging\n", servernum, readbuf);
        /* acknowledge it */
        if(retval = DosWrite(pipehandle, ACKNOWLEDGE,
            strlen(ACKNOWLEDGE)+1, &bytesread))
            error_exit(retval, "DosWrite");
        DosSleep(250L);
    }

    DosSemClear(&semhandle);
    DosClose(pipehandle);
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}
```

directory is \server\PIPE, where "server" is the name of a remote file server.

When the server process creates a named pipe, it must specify the direction in which data will flow through the pipe. If the

pipe is duplex, data flows in both directions and both processes can read from or write to the pipe. An inbound pipe is one in which the server reads data written by the client; an outbound pipe is just the opposite.

Figure B: Source Code for a Client Process

NPCLIENT.MAK

```
# make file for npclient.c
#

#COPT=/Lp /W3 /Zp /Zl /G2s /Ox /I$(INCLUDE) /Alfw
COPT=/Lp /W3 /Zp /Zl /G2s /I$(INCLUDE) /Alfw

npclient.exe: npclient.c npclient.mak
cl $(COPT) npclient.c /link /oo /noi /libcat
markexe windowcompat npclient.exe
```

NPCLIENT.C

```
/* npclient.c RHS 5/1/89
This program creates a client process which will communicate with an
instance of the named pipe created by NPSEVER. After running
NPSEVER, you can run several instances of this program with:

NPCLIENT
*/
#define INCL_DOS
#define INCL_KBD
#define INCL_ERRORS

#include<os2.h>
#include<string.h>
#include<stdio.h>
#include"nmpipe.h"

char writebuf[PIPESIZE];

void main(void);
void error_exit(USHORT err, char *msg);
PID Os2GetPid(void);

void main(void)
{
    HPIPE pipehandle;
    USHORT retval;
    USHORT bytesread;
    USHORT action;
    PID pid;

    pid = Os2GetPid();
    printf("Client %u trying to open named pipe...\n",pid);
    while(TRUE) /* open the pipe */
    {
        if(retval = DosOpen(NAMEDPIPE,&pipehandle,&action,
            0L,FILE_NORMAL,FILE_OPEN, OPEN_SHARE_DENYNONE |
            OPEN_ACCESS_READWRITE, 0L))
        {
            printf("%u: Error %u trying to open %s,
                sleeping...\n",pid,retval,NAMEDPIPE);
            if(retval == ERROR_PIPE_BUSY)
                while(TRUE)
                {
                    if(! (retval = DosWaitNmPipe(NAMEDPIPE,1000L)))
                        break;
                    if(retval != ERROR_SEM_TIMEOUT)
                        error_exit(retval,"DosWaitNmPipe");
                }
            printf("Trying again...\n");
        }
        else
            break;
    }

    printf("Client %u: server has connected pipe,
        sending message...\n", pid);

    while(TRUE)
    {
        strcpy(writebuf,"Anything I want?"); /* write a message */
        if(retval = DosWrite(pipehandle,writebuf,
            strlen(writebuf)+1,&bytesread))

```

The server process can also specify whether a pipe will transfer bytes or messages. A message is a block of data with a system-supplied header that a process can read as a unit. OS/2 leaves the size and format of the message to the discretion of the application.

Pipe States

A pipe can exist in one of several states. For example, when a server first creates a pipe, it is disconnected. After creating it, the server must place the pipe in a listening state before a client can open it. Once a client opens a listening pipe, the pipe is connected and both server and client can read from it and write to it. When the client process finishes with the pipe, it should close the pipe, placing the pipe in a closing state. Then the server process can disconnect it again.

The server must use `DosMakeNmPipe` to create the pipe initially. Then it can call `DosConnectNmPipe`, which puts the pipe in a listening state and waits until a client process opens the pipe with `DosOpen`. Once the pipe is connected, both the client and server can call `DosRead` and `DosWrite` to access the data in the pipe. When the client finishes with the named pipe, it can use `DosClose` to close the pipe. To disconnect the pipe again, the server calls `DosDisconnectNmPipe`.

Additionally, the client can use `DosCallNmPipe` to open, write, read, and close a named pipe in a single operation. If the pipe owner created the pipe with the `PIPE_WAIT` option, this function will wait until an instance of the pipe is available. The `PIPE_WAIT` option also controls whether `DosRead` and `DosWrite` wait until data or space is available in the pipe before returning. Both the client and the server can call `DosTransactNmPipe` to write to

and read from a named pipe. In addition, the client and server may call `DosPeekNmPipe` to examine data in the named pipe without removing it. Other functions allow the processes to query and set the state of a pipe and to associate the pipe with a system semaphore. OS/2 will clear the semaphore whenever data is available in the pipe. However, you cannot associate system semaphores with a remote named pipe (a named pipe that is accessed via a network file server).

The server process can use `DosMakeNmPipe` to specify how many instances of a named pipe can be opened at a time. If it specifies more than one instance, it can repeatedly call `DosMakeNmPipe` to create each additional instance of the named pipe. The server can use each instance of the pipe to service different client processes. Thus several clients can open a named pipe simultaneously.

Suppose a server process creates 10 instances of a named pipe, but all are connected (busy) when an eleventh client tries to open the pipe. `DosOpen` will return the message `ERROR_PIPE_BUSY` to this client, allowing it to call `DosWaitNmPipe`, which will block until an instance of the pipe is available. Once a named pipe instance becomes available, OS/2 will unblock the client that has waited the longest in a call to `DosWaitNmPipe`. Then the client can connect to the pipe with `DosOpen`.

If you are writing a server process that will service multiple clients simultaneously, you should create a separate thread for each instance of the pipe. Each thread should manage that instance of the pipe and attempt to connect to a client. In the event that there are more clients than there are instances of the pipe, the server thread should

Figure B

```

error_exit(retval, "DosWrite");

/* read acknowledgment */
if (retval = DosRead(pipehandle, writabuf, PIPESIZE, &bytesread))
    error_exit(retval, "DosRead");
printf("%u: Server has acknowledged: '%s'\n", pid, writabuf);
DosSleep(200L);
}
DosClose(pipehandle);
DosExit(EXIT_PROCESS, 0);
}

PID Os2GetPid(void)          /* returns process id */
{
    PIDINFO pidinfo;

    DosGetPID(&pidinfo); /* get process id */
    return pidinfo.pid; /* return it */
}

void error_exit(USHORT err, char *msg)
{
    printf("Error %u returned from %s\n", err, msg);
    DosExit(EXIT_PROCESS, 0);
}

```

Figure C: Source Code for Opening Named Pipes from DOS

NPRMODE

```

! make file for nprmode.c
!

#COPT=/W3 /Zp /Zl /G2s /Ox /I$ (INCLUDE)
COPT=/W3 /Zp1el /G2 /I$ (INCLUDE)

nprmode.exe:    nprmode.c nprmode.mak
               cl $(COPT) nprmode.c /link /co

```

NPRMODE.C

```

/* nprmode.c RRS 5/1/89
This program opens the named pipe created by NPSERVER from the DOS
compatibility environment. After starting NPSERVER in a protected
mode session, you can bring the DOS box session into the foreground
and run this program with:

NPRMODE
*/

#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<fcntl.h>
#include<io.h>
#include<share.h>
#include<sys\types.h>
#include<sys\stat.h>

#include"nmpipe.h"

#if !defined(TRUE)
#define TRUE 1
#endif

char message[80];

void main(void);

void main(void)
{
    char *mess = "Message from the DOS box!";
    int pipehandle, err, count = strlen(mess);

    if ((pipehandle = open(NAMEDPIPE, (O_BINARY | O_RDWR),

```

60

Figure C

```

    SE_DENYNO)) == -1)
    {
        printf("sopen failed, errno = %d\n", errno);
        exit(0);
    }

    while(TRUE)
    {
        strcpy(message, mess);

        if((err = write(pipehandle, message, count)) == -1)
        {
            printf("write failed, errno = %d\n", errno);
            exit(0);
        }

        count = 79;
        if((err = read(pipehandle, message, count)) == -1)
        {
            printf("read failed, errno = %d\n", errno);
            exit(0);
        }
        if(err)
        {
            message[err] = NULL;
            printf("Message received from the protected world:
                \"%s\"\n", message);
        }
        else
            printf("No message from protected world\n");
    }
    close(pipehandle);
    exit(0);
}

```

Figure D: NMPIPE.H Shared Header File

```

/* nmpipe.h RRS 5/1/89
shared header file for named pipe source */

#define NAMEDPIPE "\\PIPE\\NMPIPE"
#define QUIT_COMMAND "QUIT"
#define ACKNOWLEDGE "You got it!"
#define PIPESIZE 100

```

conduct its business with a client as efficiently as possible and disconnect as soon as it is able. Then it can connect with another client. This is not an issue if you are sure there will never be more clients than pipe instances.

Named Pipe Programs

The sample named pipe programs NPSEVER.C (see Figure A), NPCLIENT.C (see Figure B), and NPRMODE.C (see Figure C), and their common header file NMPIPE.H (see Figure D) illustrate how you can establish a server-client relationship between unrelated processes using named pipes. The server creates five server threads, but the number is arbitrary

and you can change it if you like. However, you may recall that the multithreaded library (used by these programs) only allows you to create 32 threads per process.

Each server thread opens an instance of a named pipe, waits for a client to connect, and enters a loop where it exchanges messages with a client process. The client threads open a named pipe instance and enter a loop in which they exchange messages with the server thread. Both processes print the message received from the other and sleep briefly before returning to the top of their loops. Note that if a client calls DosOpen when no instances of the pipe are available, it enters a loop in which it calls DosWaitNmPipe. Thus the client will block until a pipe instance is available.

The NPRMODE.C program is a real-mode program that can be run in the DOS compatibility environment. It illustrates how a

real-mode program that is running under DOS on a network workstation or running in the DOS-compatibility box can communicate with an OS/2 application. The program attempts to open the named pipe as if it were a file. If successful, the program repeatedly writes messages to the pipe and reads back messages passed to it by the server process. Keep in mind that real-mode programs only operate when the real-mode session is on the screen. They are frozen when you switch the real-mode session into the background.

Although I included code in both NPCLIENT.C and NPRMODE.C for closing the pipe, it's not actually reached because the programs remain in an infinite loop. If you terminate the server with Control-C, the clients will terminate when the pipe is broken. These examples are for illustrative purposes; you can insert your own code to cause a client or server to break out of the loop and close the pipe instance. More importantly, these examples should make it clear to you how easy it is to implement server software using named pipes under OS/2. □

* As used herein, "OS/2" refers to the OS/2 operating system, jointly developed by Microsoft and IBM.
 * As used herein, "DOS" refers to the MS-DOS[®] and PC-DOS operating systems.

XP002913603

The Packet Filter:

An Efficient Mechanism for User-level Network Code

Jeffrey C. Mogul

Digital Equipment Corporation Western Research Laboratory

Richard F. Rashid

Michael J. Accetta

Department of Computer Science, Carnegie-Mellon University

P.D. 1987

p. 39-51 = 13

Abstract

Code to implement network protocols can be either inside the kernel of an operating system or in user-level processes. Kernel-resident code is hard to develop, debug, and maintain, but user-level implementations typically incur significant overhead and perform poorly.

The performance of user-level network code depends on the mechanism used to demultiplex received packets. Demultiplexing in a user-level process increases the rate of context switches and system calls, resulting in poor performance. Demultiplexing in the kernel eliminates unnecessary overhead.

This paper describes the *packet filter*, a kernel-resident, protocol-independent packet demultiplexer. Individual user processes have great flexibility in selecting which packets they will receive. Protocol implementations using the packet filter perform quite well, and have been in production use for several years.

1. Introduction

It is not always appropriate to implement networking protocols inside the kernel of an operating system. Although kernel-resident network code can often outperform a user-level implementation, it is usually harder to implement and maintain, and much less portable. If optimal performance is not the primary goal of a protocol implementation, one might well prefer to implement it outside the kernel. Unfortunately, in most operating systems user-level network code is doomed to terrible performance.

In this paper we show that it is possible to get adequate performance from a user-level protocol implementation, while retaining all the features of user-level programming that make it far more pleasant than kernel programming.

The key to good performance is the mechanism used to demultiplex received packets to the appropriate user process. Demultiplexing can be done either in the kernel, or in a user-level switching process. User-mode demultiplexing allows flexible control over how packets are distributed, but is expensive because it normally involves at least two context switches and three system calls per received packet. Kernel demultiplexing is efficient, but in existing systems the criteria used to distinguish between packets are too crude.

This paper describes the *packet filter*, a facility that combines both performance and flexibility. The packet filter is part of the operating system kernel, so it delivers packets with a minimum of system calls and context switches, yet it is able to distinguish between packets according to arbitrary and dynamically variable user-specified criteria. The result is a reasonably efficient, easy-to-use abstraction for developing and running network applications.

The facility we describe is not a paper design, but the evolutionary result of much experience and tinkering. The packet filter has been in use at several sites for many years, for both development and production use in a wide variety of applications, and has insulated these applications from substantial changes in the underlying operating system. It has been of clear practical value.

In section 2, we discuss in greater detail the motivation for the packet filter. We describe the abstract interface in section 3, and briefly sketch the implementation in section 4. We then illustrate, in section 5, some uses to which the packet filter has been put, and in section 6 discuss its performance.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 089791-242-X/87/0011/0039 \$1.50

2. Motivation

Software to support networking protocols has become tremendously important as a result of use of LAN technology and workstations. The sheer bulk of this software is an indication that it may be overwhelming our ability to create reliable, efficient code: for example, 30% of the 4.3BSD Unix [8, 21] kernel source, 25% of the TOPS-20 [10] (Version 6.1) kernel source, and 32% of the V-system [4] kernel source are devoted to networking.

Development of network software is slow and seldom yields finished systems; debugging may continue long after the software is put into operation. Continual debugging of production code results not only from deficiencies in the original code, but also from inevitable evolution of the protocols and changes in the network environment.

In many operating systems, network code resides in the kernel. This makes it much harder to write and debug:

- Each time a bug is found, the kernel must be recompiled and rebooted.
- Bugs in kernel code are likely to cause system crashes.
- Functionally independent kernel modules may have complex interactions over shared resources.
- Kernel-code debugging cannot be done during normal timesharing; single-user time must be scheduled, resulting in inconvenience for timesharing users and odd work hours for system programmers.
- Sophisticated debugging and monitoring facilities available for developing user-level programs may not be available for developing kernel code.
- Kernel source code is not always available.

In spite of these drawbacks, network code is still usually put in the kernel because the drawbacks of putting it outside the kernel seem worse. If a single user-level process is used for demultiplexing packets, then for each received packet the system will have to switch into the demultiplexing process, notify that process of the packet, then switch again as the demultiplexing process transfers the packet to the appropriate destination process. (Figure 2-1 depicts the costs associated with this approach.) Context switching and inter-process communication are usually expensive, so clearly it would be more efficient to immediately deliver each packet to the ultimate destination process. (Figure 2-2 shows how this approach reduces costs.) This requires that the kernel be able to determine to which process each packet should go; the problem is how to allow user-level processes to specify which packets they want.

One simple mechanism is for the kernel to use a specific packet field as a key; a user process registers with the kernel the field value for packets it wants to

receive. Since the kernel does not know the structure of higher-layer protocol headers, the discriminant field must be in the lowest layer, such as an Ethernet [9] "type" field. This is not always a good solution. For example, in most environments the Ethernet type field serves only to identify one of a small set of protocol families; almost all packets must be further discriminated by some protocol-specific field. If the kernel can demultiplex only on the type field, then one must still use a user-level switching process with its attendant high cost.

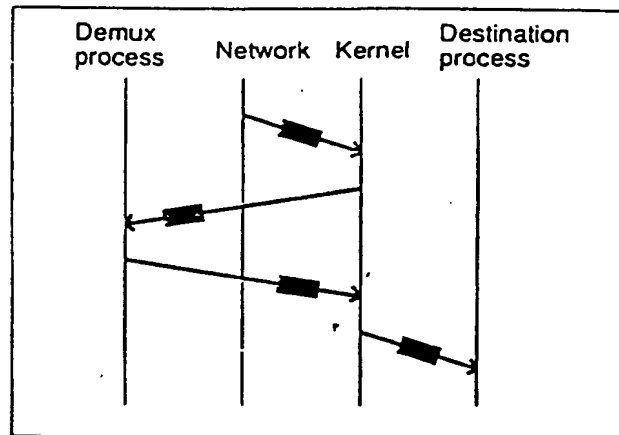


Figure 2-1: Costs of demultiplexing in a user process

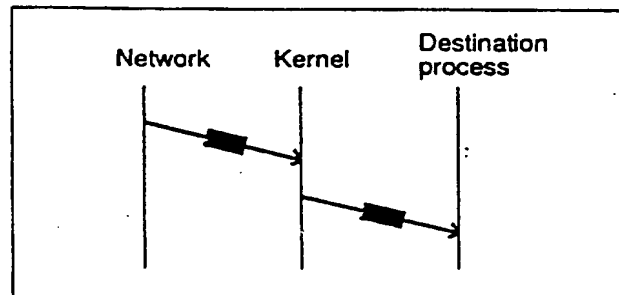


Figure 2-2: Costs of demultiplexing in the kernel

The packet filter is a more flexible kernel-resident demultiplexer. A user process specifies an arbitrary predicate to select the packets it wants; all protocol-specific knowledge is in the program that receives the packets. There is no need to modify the kernel to support a new protocol. This mechanism evolved for use with Ethernet data-link layers, but will work with most similar datagram networks.

The packet filter not only isolates the kernel from the details of specific protocols; it insulates protocol code from the details of the kernel implementation. The packet filter is not strongly tied to a particular system; in its Unix implementation, it is cleanly separated from other kernel facilities and the novel part of the user-level interface is not specific to Unix. Because protocol code lives outside the kernel it does not have to be

modified to be useful with a wide variety of kernel implementations. In systems where context-switching is inexpensive, the performance advantage of kernel demultiplexing will be reduced, but the packet filter may still be a good model for a user-level demultiplexer to emulate.

In addition to the cost and inconvenience of demultiplexing, the cost of domain crossing whenever control crosses between kernel and user domains has discouraged the implementation of protocol code in user processes. In many protocols, far more packets are exchanged at lower levels than are seen at higher levels (these include control, acknowledgement, and duplicate packets). A kernel-resident implementation confines these overhead packets to the kernel and greatly reduces domain crossing, as depicted in figure 2-3. The packet filter mechanism cannot eliminate this problem; we can reduce it through careful implementation and by batching together domain-crossing events (see section 3).

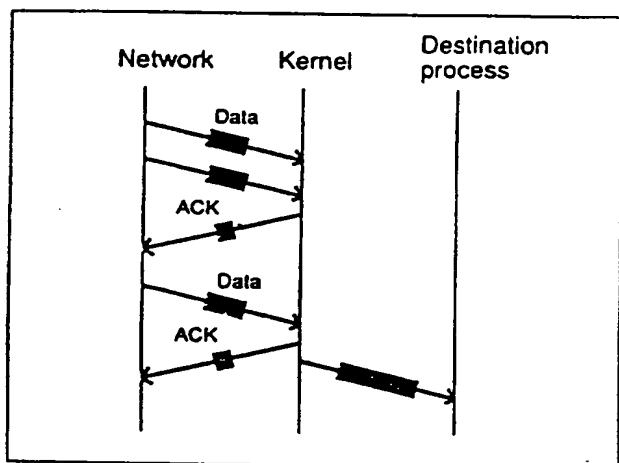


Figure 2-3: Kernel-resident protocols reduce domain-crossing

User-level access to the data-link layer is not universally regarded as a good thing. Some have suggested that user programs never need access to explicit network communication [23]; others might argue that all networking should be done within a transport protocol such as IP [19] or the ISO Transport Protocol [15], with demultiplexing done by the transport layer code. Both these arguments implicitly assume a homogeneous networking environment, but heterogeneity is often a fact of life: machines from different manufacturers speak various transport protocols, and research on new protocol designs at the data-link level is still profitable.

The packet filter allows rapid development of networking programs, by relatively inexperienced programmers, without disrupting other users of a timesharing system. It places few constraints on the protocols that may be implemented, but in spite of this flexibility it performs well enough for many uses.

2.1. Historical background.

As far as we are aware, the idea (and name) of the packet filter first arose in 1976, in the Xerox Alto [3]. Because the Alto operating system shared a single address space with all processes, and because security was not important, the filters were simply procedures in the user-level programs; these procedures were called by the packet demultiplexing mechanism. The first Unix implementation of the packet filter was done in 1980.

3. User-level interface abstraction

Figure 3-1 shows how the packet filter is related to other parts of a system. Packets received from the network are passed through the packet filter and distributed to user processes; code to implement protocols lives in each process. Figure 3-2 shows, for contrast, how networking is done in "vanilla" 4.3BSD Unix; protocols are implemented inside the kernel and data buffers are passed from protocol code to user processes. Figure 3-3 shows how both models can coexist; some programs may even use both means to access the network.

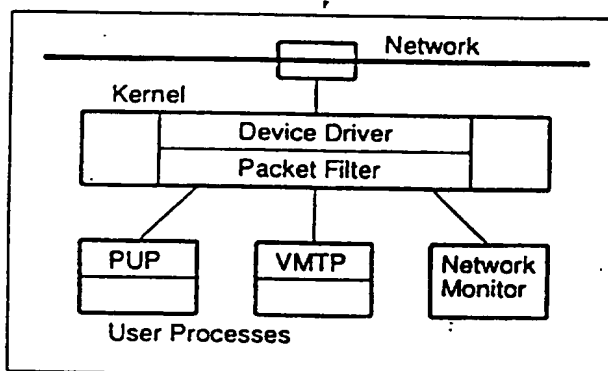


Figure 3-1: Relationship between packet filter and other system components

The programmer's interface to the packet filter has three major components: packet transmission, packet reception, and control and status information. We describe these in the context of the 4.3BSD Unix implementation.

Packet transmission is simple; the user presents a buffer containing a complete packet, including data-link header, to the kernel using the normal Unix *write* system call; control returns to the user once the packet is queued for transmission. Transmission is unreliable if the data link is unreliable; the user must discover transmission failure through lack of response rather than an explicit error.

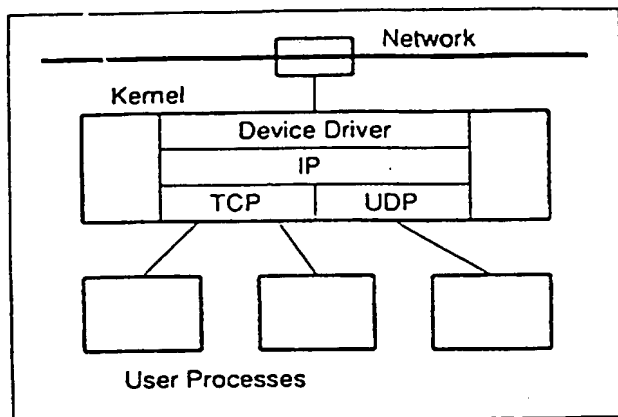


Figure 3-2: 4.3BSD networking model

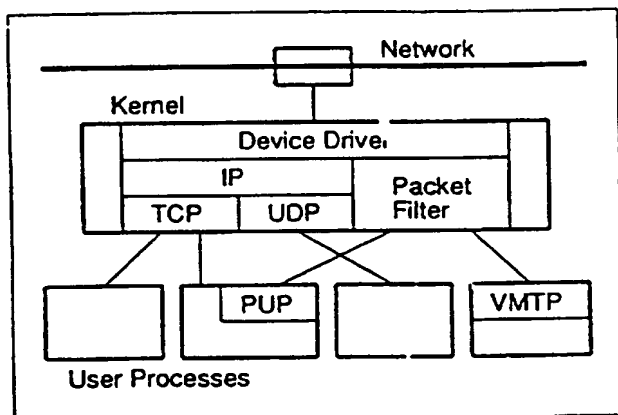


Figure 3-3: Packet filter coexisting with 4.3BSD networking model

Packet reception is more complicated. The packet filter manages some number of *ports*, each of which may be opened by a Unix program as a "character special device." Associated with each port is a *filter*, a user-specified predicate on received packets. If a filter accepts a packet, the packet is queued for delivery to the associated port. A filter is specified using a small stack-based "language," in which one can push arbitrary constants or words from the received packet, and apply binary operations to the top of the stack. The filter language is discussed in more detail in section 3.1. A process binds a filter to a port using an *ioctl* system call; a new filter can be bound at any time, at a cost comparable to that of receiving a packet; in practice, filters are not replaced very often.

Two processes implementing different communication streams under the same protocol must specify slightly different predicates so that packets are delivered appropriately. For example, a program implementing a Pup [2] protocol would include a test on the Pup destination socket number as part of its predicate. The layering in a protocol architecture is not necessarily reflected in a filter predicate, which may well examine packet fields from several layers.

When a program performs a *read* system call on the file descriptor corresponding to a packet filter port, the first of any queued packets is returned. The entire packet, including the data-link layer header, is returned, so that user programs may implement protocols that depend on header information. The program may ask that all pending packets be returned in a batch; this is useful for high-volume communications because it can amortize the overhead of performing a system call over several packets. Figure 3-4 depicts per-packet overheads without batching; figure 3-5 shows how these are reduced when batching is used.

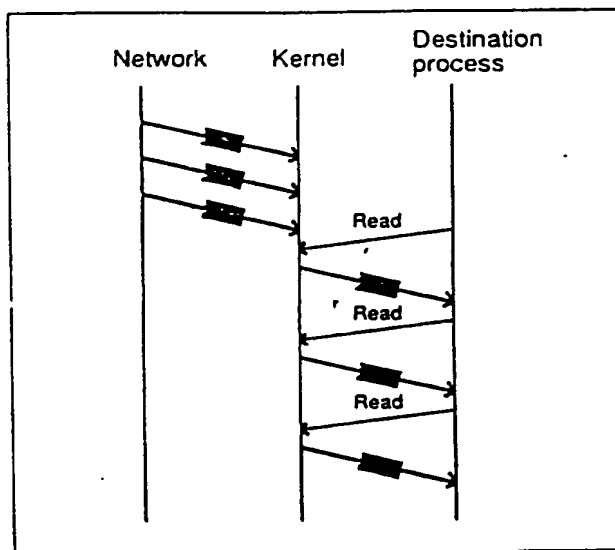


Figure 3-4: Delivery without received-packet batching

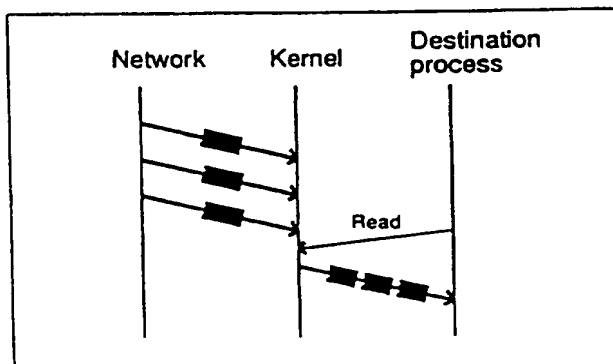


Figure 3-5: Delivery with received-packet batching

If no packets are queued, the *read* system call blocks until a packet is available; if no packet arrives during a timeout period, the *read* call terminates and reports an error. Simple programs can be written using a "write; read with timeout; retry if necessary" paradigm. More elaborate programs may take advantage of two more sophisticated synchronization mechanisms: the 4.3BSD *select* system call, or a interrupt-like facility using Unix

"signals," either of which allows non-blocking network I/O.

3.1. Filter language details

The heart of the packet filter is an interpreter for the "language" shown in figure 3-6. A filter is a data structure including an array of 16-bit words. Each word is normally interpreted as an instruction with two fields, a *stack action* field and a *binary operation* field.

A stack action may cause either a word from the received packet or a constant to be pushed on the stack. A binary operation pops the top two words from the stack, and pushes a result. Thus, filter programs evaluate a logical expression composed of tests on the values of various fields in the received packet. The filter is normally evaluated until the program is exhausted. If the value remaining on top of the stack is non-zero, the filter is deemed to have accepted the packet.

It is sometimes possible to avoid evaluating the entire filter before deciding whether to accept a packet. This is especially important for performance, since on a busy system several dozen filters may be applied to an incoming packet before it is accepted. The filter language therefore includes four "short-circuit" binary logical operations, that when evaluated either push a result and allow the program to continue, or terminate the program and return an appropriate boolean.

Figure 3-8 shows an example of a simple filter program; figure 3-9 shows an example of a filter program using short-circuit operations. Both are used with Pup [2] packets on a 3Mbit/sec. Experimental Ethernet [17]; the data-link header is 4 bytes (two words) long, with the packet type in the second word (see figure 3-7.) In normal use, the filters are not directly constructed by the programmer, but are "compiled" at run time by a library procedure.

The design of the filter language is not the result of careful analysis but rather embodies several accidents of history, such as its bias towards 16-bit fields. It has evolved over the years; in particular, the short-circuit operations were added after an analysis showed that they would reduce the cost of interpreting filter predicates. One could imagine alternatives to the stack language structure; for example, a predicate could be an array of (*field-offset*, *expected-value*) pairs, and the predicate would be satisfied if all the specified fields had the specified values. However, the additional flexibility of the stack language has often proved useful in constructing efficient filters. Since the "instruction set" is implemented in software, not hardware, there is no execution-time penalty associated with supporting a broad range of operations.

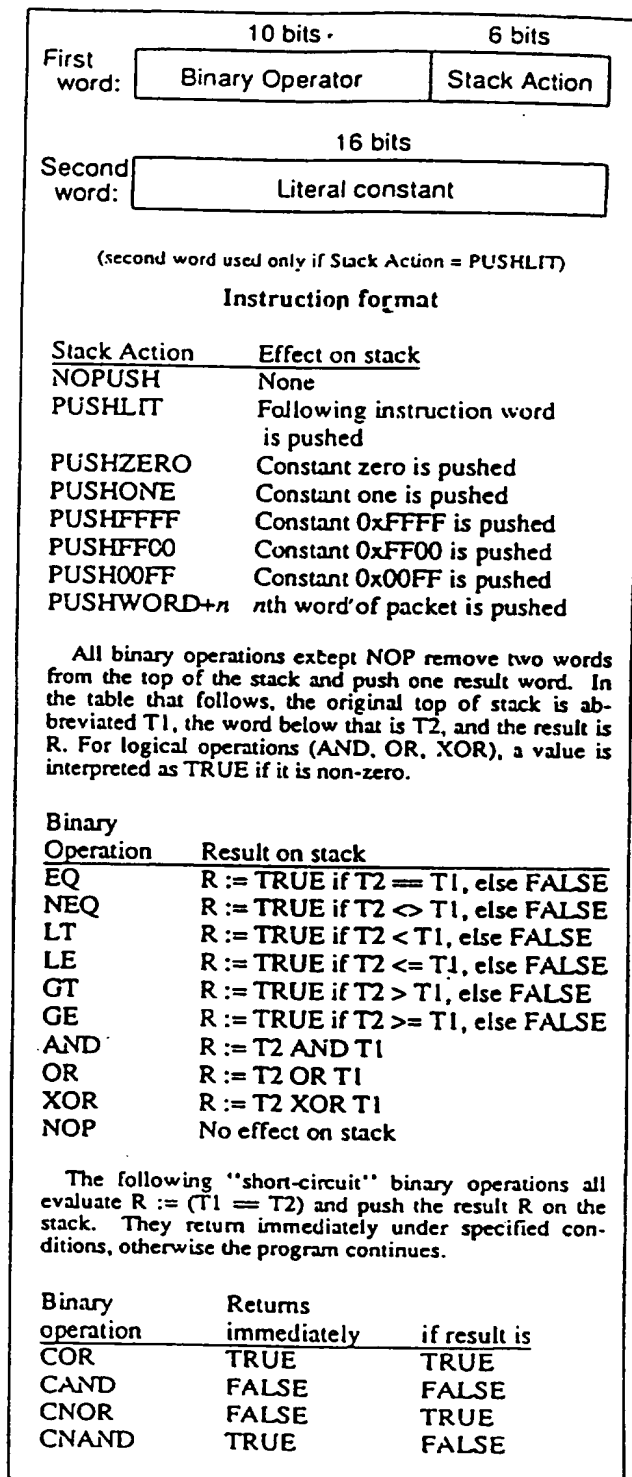


Figure 3-6: Packet filter language summary

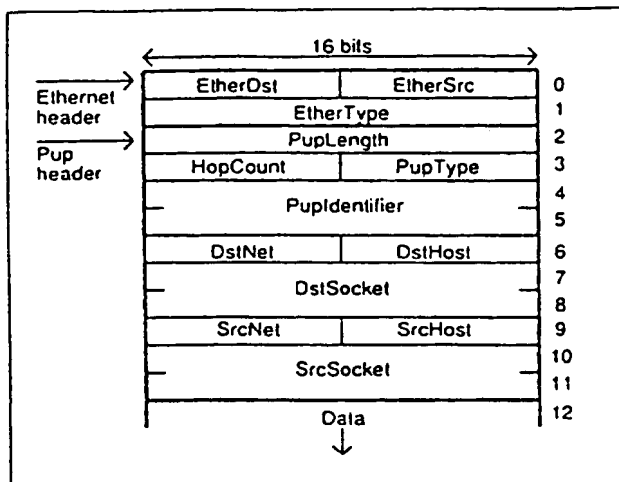


Figure 3-7: Format of Pup Packet header on JMB Ethernet (after [2])

This filter accepts all Pup packets with Pup Types between 1 and 100. The Pup Type field is a one byte field, so it must be masked out of the appropriate word in the packet.

```
struct enfilter f = {
    10, 12, /* priority and length */
    PUSHWORD+1, PUSHLIT | EQ, 2,
    /* packet type == PUP */
    PUSHWORD+3, PUSH00FF | AND,
    /* mask low byte */
    PUSHZERO | GT,
    /* PupType > 0 */
    PUSHWORD+3, PUSH00FF | AND,
    /* mask low byte */
    PUSHLIT | LE, 100,
    /* PupType <= 100 */
    AND, /* 0 < PupType <= 100 */
    AND /* && packet type == PUP */
};
```

Figure 3-8: Example filter program

This filter accepts Pup packets with a Pup DstSocket field of 35. The DstSocket field occupies two words, so the filter must test both words and combine them with an AND operation. The DstSocket field is checked before the packet type field, since in most packets the DstSocket is likely not to match and so the short-circuit operation will exit immediately.

```
struct enfilter f = {
    10, 8, /* priority and length */
    PUSHWORD+8, PUSHLIT | CAND, 35,
    /* Low word of socket == 35 */
    PUSHWORD+7, PUSHZERO | CAND,
    /* High word of socket == 0 */
    PUSHWORD+1, PUSHLIT | EQ, 2
    /* packet type == Pup */
};
```

Figure 3-9: Example filter program using short-circuit operations

3.2. Access Control

Normally, once a packet has been accepted for delivery to a process, it will not be submitted to the filters of any other processes. Because it is not possible to determine if two filters will accept overlapping sets of packets, we need a way to prevent one process from inappropriately diverting packets meant for another process.

Associated with each filter is a *priority*; filters are applied in order of decreasing priority, so if two filters would both accept a packet, it goes to the one with higher priority. (Priority has another purpose: if priorities are assigned proportional to the likelihood that a filter will accept a packet, then the "average" packet will match one of the first few filters it is tested against, consequently reducing the amount of filter interpretation overhead.) If two filters have the same priority, the order of application is unspecified (the interpreter may occasionally reorder such filters to place the busier ones first); in these cases one must take care to ensure that the filters accept disjoint sets of packets.

Optionally, a process may specify that the packets accepted by its filter should be submitted to other, lower-priority, filters as well; multiple copies of such packets may be delivered. This is useful in implementing monitoring facilities without disturbing the processes being monitored, in "group" communication where a packet may be multicast to several processes on one host, or when it is not possible to filter precisely enough within the kernel.

This access control mechanism does not in itself protect against malicious or erroneous processes attempting to divert packets; it only works when processes play by the rules. In the research environment for which the packet filter was developed, this has not been a problem, especially since there are many other ways to eavesdrop on an Ethernet. An earlier version of the packet filter did provide some security by restricting the use of high-priority filters to certain users, allowing these users first rights to all packets, but this mechanism went unused.

Because typical networks are easily tapped, most proposals for secure communication rely on encryption to protect against eavesdropping. If packets are encrypted, some header fields must be transmitted in cleartext to allow demultiplexing; this is not peculiar to use of the packet filter, especially if encryption is on a per-process basis.

3.3. Control and status information

The user can control the packet filter's action in a variety of ways, by specifying: the filter to be associated with a packet filter port; the timeout duration for blocking reads (or optionally, immediate return or indefinite blocking); the signal, if any, to be delivered upon packet reception; and the maximum length of the per-port input queue.

Information provided by the packet filter to programs includes: the type of the underlying data-link layer; the lengths of a data-link layer address and of a data-link layer header; the maximum packet size for the data-link; the data-link address for incoming packets; and the address used for data-link layer broadcasts, if one exists.

The user can also ask that each received packet be marked with a timestamp and a count of the number of packets lost due to queue overflows in the network interface and in the kernel.

4. Implementation

The packet filter is implemented in 4.3BSD Unix as a "character special device" driver. Just as the Unix terminal driver is layered above communications device drivers to provide a uniform abstraction, the packet filter is layered above network interface device drivers. As with any character device driver, it is called from user code via *open*, *close*, *read*, *write*, and *ioctl* system calls. The packet filter is called from the network interface drivers upon receipt of packets not destined for kernel-resident protocols.

Most of the complexity in the implementation is involved in bookkeeping and in managing asynchrony. When a packet is received, it is checked against each filter, in order of decreasing priority, until it is accepted or until all filters have rejected it (see figure 4-1). The filter interpreter is straightforward, but must be carefully coded since its inner loop is quite busy. It simply iterates through the "instruction words" of a filter (there are no branch instructions), evaluating the filter predicate using a small stack. When it reaches the end of the filter, or a short-circuit conditional is satisfied, or an error is detected, it returns the predicate value to indicate acceptance or rejection of the packet.

```
Accepted := false;
for priority := MaxPriority downto
    MinPriority do
    for i := FirstFilter[priority] to
        LastFilter[priority] do
        if Apply(Filter[i], rcvd_pkt)
            = MATCH then
            Deliver(Port[i], rcvd_pkt);
            Accepted := true;
        end;
    end;
end;
if not Accepted then
    Drop(rcvd_pkt);
end;
```

Figure 4-1: Pseudo-code for filter application loop

The packet filter module is about 2000 lines of heavily-commented C source code (under 6K bytes of Vax machine code); each of the network interface device drivers must be modified with a few dozen lines of linkage code. Aside from this, the packet filter requires no modification of the Unix kernel. Because it is well-isolated from the rest of the kernel, it is easily ported to different Unix implementations. Ports have been made to the Sun Microsystems Inc. operating system, which is internally quite similar to 4.2BSD, and to the Ridge Operating System (ROS) of Ridge Computers, Inc. ROS is a message-based operating system with inexpensive processes [1]; its internal structure is distinctly different from that of Unix. The packet filter has also been ported to Pyramid Technology's Unix system, with minor modification for use in a multi-processor. It appears to be relatively easy to port the packet filter to a variety of operating systems; this in turn makes it possible to port user-level networking code without further kernel modifications.

5. Uses of the packet filter

The packet filter is successful because it provides a useful facility with adequate performance. Section 6 provides quantitative measures of performance; in this section we consider qualitative utility.

The primary goal of the packet filter is to simplify the development and improvement of networking software and protocols. Since networking software is often in a continual state of development, anything that speeds debugging and modification reduces the mismatch between the software and the networking environment. This is especially important for the experimental development of new protocols. Similarly, since operating systems are continually changing, decoupling network code from the rest of the system reduces the risk of "software rot."

The remainder of this section describes examples demonstrating how the packet filter has been of practical use.

5.1. Pup protocols

The Pup [2] protocol suite includes a variety of applications using both datagram (request-response) and stream transport protocols. At Stanford, almost all of the Pup protocols were implemented for Unix, based entirely on the packet filter. Although Pup, as an experimental architecture, has some notable flaws, for about five years this implementation served as the primary link between Stanford's Unix systems and other campus hosts and workstations. Pup is still in relatively heavy use in a number of organizations, most of which have used the Stanford implementation.

The experience with Pup has shown the value of decoupling the networking implementation from the Unix kernel. Not only did this make it possible to develop the Pup code without the effort of kernel

debugging, it also made it possible to modify the kernel without having to worry about the integrity of the Pup code. When, every few years, a new release of the Berkeley Unix kernel became available, it sufficed to re-install the kernel module implementing the packet filter. The Pup code could then be run, often without recompilation, under the new operating system. The initial port of the packet filter code from 4.1BSD to 4.2BSD took several evenings; for comparison, it took six programmer-months to port BBN's TCP implementation from 4.1BSD to 4.2BSD [14]. That the BBN TCP code is kernel-resident undoubtedly contributed to the time it took to port.

5.2. V-system protocols

The V-system is a message-based distributed operating system. As an ongoing research project, it is under continual development and revision. The architects of the V-system have chosen to design their own protocols, to obtain high performance and so that they could make use of the multicast feature of Ethernet hardware [6].

Although the V-system is primarily a collection of workstations and servers running the V kernel, Unix hosts were integrated into the distributed system to provide disk storage, compute cycles, mail service, and other amenities not available in a new operating system. The Unix hosts had to be taught to speak the V-system Inter-Kernel Protocol (IKP). Fortunately, the packet filter was available for use as the basis of a user-level V IKP server process.

The V IKP is a simple protocol and could have been put in the Unix kernel. This, however, would have required the V researchers to learn about the details of the Unix kernel, to participate in the maintenance of the kernel, and to re-install the IKP implementation in each new release of the operating system. Instead, they were able to devote their attention to research on the topics that interested them. One result of this research was the VMTP protocol [5], a replacement for the V IKP. Although there is a kernel-resident implementation of VMTP for 4.3BSD, the first implementation used the packet filter. The user-level implementation allowed rapid development of the protocol specification through experimentation with easily-modified code. (Section 6.3 contrasts the performance differences between the two VMTP implementations.)

5.3. RARP

The Reverse Address Resolution Protocol (RARP) [12] was designed to allow workstations to determine their Internet Protocol (IP) addresses without relying on any local stable storage. One issue in the definition of this protocol was whether it should be a layer above IP, or a parallel layer. The former leads to a chicken-or-egg dilemma; the latter is cleaner but raised question of implementability under 4.2BSD. With the packet filter, however, a RARP implemen-

tation was easy; the work was done in a few weeks by a student who had no experience with network programming, and who had no need to learn how to modify the Unix kernel.

5.4. Network Monitoring

For the developer or maintainer of network software, no tool is as valuable as a *network monitor*. A network monitor captures and displays traces of the packets flowing among hosts; a packet trace makes it much easier to understand why two hosts are unable to communicate, or why performance is not up to par.

Most commercially-available network monitors (including the Excelan *LANalyzer* [11], the Network General *Sniffer* [18], and the Communications Machinery Corp. *LanScan* [7]) are stand-alone units dedicated to monitoring specific protocols. A network monitor closely integrated with a general-purpose operating system, running on a workstation, has several important advantages over a dedicated monitor:

- All the tools of the workstation are available for manipulating and analyzing packet traces.
- A user can write new monitoring programs to display data in novel ways, or to monitor new or unusual protocols.

One of us has been using the packet filter, on a MicroVAX-II workstation, as the basis for a variety of experimental network monitoring tools. This system has sufficient performance to record all packets flowing on a moderately busy Ethernet (with rare lapses), and more than sufficient performance to capture all packets between a pair of communicating hosts. Since one can easily write arbitrarily elaborate programs to analyze the trace data, and even to do substantial analysis in real time, an integrated network monitor appears to be far more useful than a dedicated one. (Sun Microsystems' *etherfind* program is another example of an integrated network monitor. It is based on Sun's *Network Interface Tap* (NIT) facility, which is similar to the packet filter but only allows filtering on a single packet field¹ [22].)

6. Performance

We measured the performance of the packet filter in several ways. We determined the amount of processor time spent on packet filter routines, and we measured the throughput of protocol implementations based on the packet filter. We compared these measurements with those for kernel-resident implementations of similar protocols, and found that in practice packet-filter-based protocol implementations perform fairly well.

All measurements were made using VAX processors

¹Sun expects to include our packet-filtering mechanism in a future release of NIT.

running 4.2BSD or 4.3BSD Unix, using either a 10Mbit/sec or 3Mbit/sec Ethernet. Note that the packet filter coexists with kernel-resident protocol implementations, without affecting their performance.

6.1. Kernel per-packet processing time

One indication of the packet filter's cost is the kernel CPU time required to process an "average" received packet. We measured this time for the packet filter, and for analogous functions of kernel-resident protocols. A 4.3BSD Unix kernel was configured to collect the CPU time spent in and number of calls made to each kernel subroutine. The profiled kernel was run for 28 hours on a timesharing VAX-11/780, and *gprof* [13] was used to format the data.

During the profiling period, the system handled 1.3 million packets. 21% of these packets were processed by the packet filter; of the remainder, 69% were IP packets and 10% were ARP packets. All per-packet processing times reported are for "average" packets and "typical" filter predicates.

Processing times for transmitted packets are about the same for either the packet filter or the kernel-resident IP implementation; it takes about 1 mSec to send a datagram. The packet filter has a slight edge, since it does not need to choose a route for the datagram or compute a checksum.

Packet filter: The packet filter spends an average of 1.57 mSec processing each packet. 41% of this time is spent evaluating filter predicates; the average packet is tested against 6.3 predicates. We derived a crude estimate for the time to process a packet: $0.8 \text{ mSec} + (0.122 * \text{number of predicates tested}) \text{ mSec}$. The average number of predicates tested will normally be somewhat less than half the number of active ports, because the priority mechanism described in section 3.2 can cause the most likely filters to be tested first.

Kernel-resident IP implementation: The average time required to process a received IP packet was 1.77 mSec. This includes all protocol processing up to the TCP and UDP layers; if only the IP layer processing is counted, the average packet requires about 0.49 mSec. This means that the kernel-resident IP layer is about three times faster than the packet filter at processing an average packet.

6.2. Total per-packet processing time

The kernel profile does not account for the entire cost of handling packets. We measured actual packet rates into and out of user processes on a microVax-II running Ultrix 1.2, using a synthetic load. The results for packet reception are included in tables 6-8 and 6-9 in section 6.5.

Although sending datagrams via the packet filter costs less than sending an unchecksummed UDP

datagram of the same size (see table 6-1), we estimate that this is still about twice the cost for the kernel to send a datagram on its own. For packets that carry no useful data (acknowledgements, for example) user-level protocol implementations pay this additional penalty.

Total packet size	Elapsed time per packet sent via packet filter	via UDP
128 bytes	1.9 mSec	3.1 mSec
1500 bytes	3.6 mSec	4.9 mSec

Table 6-1: Cost of sending packets

6.3. VMTP performance

The only interesting protocol for which there is both a packet-filter based implementation and a kernel-resident implementation is VMTP [5]. This provides a basis for a direct measurement of the cost of user-level implementation; while there are minor differences in the actual protocols implemented, and the two implementations are not of precisely equal quality, they follow essentially the same pattern of packet transport. All these measurements, unless noted, were carried out using microVax-II processors, 4.3BSD Unix, and a 10Mbit/sec Ethernet. In each case, both ends of the transfer used identical protocol implementations.

We measured the cost for a minimal round-trip operation (reading zero bytes from a file). The results, shown in table 6-2, indicate that the penalty for user-level implementation is almost exactly a factor of two. On this measurement, the Unix kernel implementation of VMTP is quite close to the V kernel implementation, indicating that there is no obvious inefficiency in the Unix kernel implementation.

VMTP Implementation	elapsed time/operation
Packet filter	14.7 mSec
Unix kernel	7.44 mSec
V kernel	7.32 mSec

Table 6-2: Relative performance of VMTP for small messages

We also measured the cost for transferring bulk data using VMTP. This was done by repeatedly reading the same segment of a file, which therefore stayed in the file system buffer cache; consequently, the measured rates should be nearly independent of disk I/O speed. (In each trial about 1 Mb was transferred.) We also measured TCP performance, for comparison; note that TCP checksums all data, whereas these implementations of VMTP do not. The results, shown in table 6-3, show that in this case the penalty for user-level

implementation of VMTP is almost exactly a factor of three.

Implementation	Rate
Packet filter	112 Kbytes/sec
Unix kernel VMTP	336 Kbytes/sec
V kernel VMTP	278 Kbytes/sec
Unix kernel TCP	222 Kbytes/sec

Table 6-3: Relative performance of VMTP for bulk data transfer

The packet-filter based implementation measured in table 6-3 uses received-packet batching. Table 6-4 shows that batching improves throughput by about 75% over identical code that reads just one packet per system call; the difference cannot be entirely due to decreased system call overhead, but may reflect reductions in context switching and dropped packets.

Batching	Rate
Yes	112 Kbytes/sec
No	64 Kbytes/sec

Table 6-4: Effect of received-packet batching on performance

We also tried to measure the cost of a user-level demultiplexing process, by simulating it within the client VTMP implementation. This is done by using an extra process to receive packets, which are then passed to the actual VMTP process via a Unix pipe. (In this case, the server process was *not* modified.) Table 6-5 shows that user-level demultiplexing has a small cost (20% greater latency) for short messages, but decreases bulk throughput by more than a factor of four (much of this is attributable to the poor IPC facilities in 4.3BSD).

Demultiplexing done in	Elapsed time per minimal operation	Bulk rate
Kernel	14.72	112 Kbytes/sec
User process	18.08	25 Kbytes/sec

Table 6-5: Effect of user-level demultiplexing on performance

6.4. Byte-stream throughput

We compared the performance of a Pup/BSP (Byte-Stream Protocol) implementation using the packet filter with that of the IP/TCP [20] implementation in the 4.3BSD kernel. These measurements were carried out using microVax-II processors, 4.3BSD Unix, and a 10Mbit/sec Ethernet.

Table 6-6 shows the rates at which the two implementations can transfer bulk data from process to process. TCP is faster by almost a factor of six. When used to implement a File Transfer Protocol (FTP), TCP slows by a factor of two if the source of data is a disk file, but the BSP throughput remains unchanged, indicating that network performance is the rate-limiting factor for BSP file transfer.

Implementation	Rate
Packet filter BSP	38 Kbytes/sec
Unix kernel TCP	222 Kbytes/sec

Table 6-6: Relative performance of stream protocol implementations

Pup (hence BSP) allows a maximum packet size of 568 bytes, whereas TCP in 4.3BSD uses 1078-byte packets and so sends only half as many; we found that if TCP is forced to use the smaller packet size, its performance is cut in half. After this correction, TCP throughput is still three times that of BSP; most of difference is attributable to the cost of BSP's user-level implementation. This is consistent with the factor-of-two difference we measured for VMTP.

We also measured performance for Telnet (remote terminal access)². A program on the "server" host (Vax-11/780) prints characters which are transmitted across the network and displayed at the "user" host. The results are shown in table 6-7. The "Output rate" column shows the overall throughput, in characters per second, for each configuration.

Telnet protocol	Network bandwidth	Output rate
Pup/BSP	10 Mbit/sec	1635
IP/TCP	10 Mbit/sec	1757
Pup/BSP	3 Mbit/sec	878
IP/TCP	3 Mbit/sec	933

Table 6-7: Relative performance of Telnet

The first two rows of the table show throughput using an MC68010-based workstation capable of displaying about 3350 characters per second. The achieved throughput is about half that, varying only slightly according to whether TCP or BSP (and thus the packet filter) is used. The last two rows, measured with characters displayed on a 9600 baud terminal, show almost no difference between BSP and TCP performance. These output rates are clearly limited by the display terminal, not by network performance.

²This test was done under 4.2BSD.

In summary, a kernel-resident implementation of a stream protocol such as VMTP or BSP appears to be about two or three times as fast as an implementation based on the packet filter. In many applications, the actual performance difference may be much smaller; the packet-filter implementation of VMTP is only 40% slower than the kernel-resident TCP when used for file transfer. The VMTP and BSP implementations are quite useful in practice; disks and terminals are more often serious bottlenecks than the packet filter.

6.5. Costs of demultiplexing outside the kernel

We asserted in section 2 that using a user-level process to demultiplex received packets to other processes would result in poor performance. In section 6.3 we showed that this appears to be true, especially for bulk-data transfer. In this section, we analyze the additional cost using measurements of Ultrix 1.2; the measurements are inspired by those made by McKusick, Karels, and Leffler [16].

6.5.1. Analytical model

If a demultiplexing process is used, each received packet results in at least two context switches: one into the demultiplexing process and one into the receiving process³. If the system has other active processes, an additional context switch to an unrelated process may occur, when the receiving process blocks waiting for the next packet.

With direct delivery of received packets, in the best case the receiving process will never be suspended, and no context switches take place. In the worst case, with other active processes, a received packet will cause two context switches.

Either mechanism requires at least one data transfer between kernel and process. Since Unix does not support memory sharing, the demultiplexing process requires two additional data transfers to get the packet into the final receiving process.

6.5.2. Cost of overhead operations

Benchmarks indicate that a MicroVAX-II running Ultrix 1.2 requires about 0.4 mSec of CPU time to switch between processes, and about 0.5 mSec of CPU time to transfer a short packet between the kernel and a process. Therefore, we predict that receiving a short packet using a demultiplexing process should take at least 2.3 mSec while for the packet filter, these overhead costs may be as low as 0.5 mSec per packet; the difference increases for longer packets because data copying requires about 1 mSec/Kbyte.

³We assume that no batching of packets takes place; this assumption breaks down when packets arrive faster than the system can switch contexts.

6.5.3. Measured costs

These costs are not the only ones associated with receiving a packet; they are the ones that are affected by the use of user-level demultiplexing. We measured the actual elapsed time required to receive packets of various sizes; the "demultiplexing process" receives packets from the network and passes them to a second process via a Unix pipe. The results are shown in table 6-8. The additional cost of user-level demultiplexing agrees fairly closely with our predication.

Packet size	Elapsed time if demultiplexing done in kernel	done in user process
128 bytes	2.3 mSec	5.0 mSec
1500 bytes	4.0 mSec	9.0 mSec

Table 6-8: Per-packet cost of user-level demultiplexing

Since received-packet batching, as we saw in section 6.3, can amortize the costs of context-switching over many packets, we repeated our measurements with batching enabled; the batch size was hard to control but the results are about the same for four or more packets per batch. The results are shown in table 6-9; batching clearly reduces the penalty associated with user-level demultiplexing, but the difference remains significant.

Packet size	Elapsed time if demultiplexing done in kernel	done in user process
128 bytes	1.9 mSec	2.4 mSec
1500 bytes	3.5 mSec	5.9 mSec

Table 6-9: Per-packet cost of user-level demultiplexing with received-packet batching

The measurements in tables 6-8 and 6-9 were made without any real decision-making on the part of the demultiplexer. Before we condemn user-level demultiplexing on the basis of its high overhead, we must show that the cost of interpreting packet filters in the kernel does not dwarf the benefit of avoiding context switches (presumably, a user-level demultiplexer could make decisions at least as efficiently and possibly more so). We measured the cost of interpreting filter programs of various lengths; the results are shown in table 6-10. (Batching was enabled and all packets were 128 bytes long.) It usually takes two or three filter instructions to test one packet field; even with rather long filters (21 instructions) the additional cost for filter interpretation is less than the cost of user-level demultiplexing if no more than three such long filters are applied to an incoming packet before one filter accepts it.

Filter length (instructions)	Elapsed time per packet
0	1.9 mSec
1	2.0 mSec
9	2.2 mSec
21	2.5 mSec

Table 6-10: Cost of interpreting packet filters

For filters using short-circuit conditionals, the break-even point is closer to an average of about ten filters before acceptance, which should occur when more than twenty filters are active. This means that even if one assumes zero cost for decision-making in a user-level demultiplexer, the break-even point comes with twenty different processes using the network. For packets longer than 128 bytes, the break-even point comes with even more active processes.

In summary, kernel demultiplexing performs significantly better than user-level demultiplexing for a wide range of situations. This advantage disappears only if a very large number of processes are receiving packets.

7. Problems and possible improvements

Since its beginnings in early 1980, the packet filter has often been revised to support additional applications or provide better performance. There is still room for improvement.

The filter language described in section 3 only allows the user to specify packet fields at constant offsets from the beginning of a packet. This has been adequate for protocols with fixed-format headers (such as Pup), but many network protocols allow variable-format headers. For example, since the IP header may include optional fields, fields in higher layer protocol headers are not at constant offsets. The current packet filter can be made to handle non-constant offsets only with considerable awkwardness and inefficiency; the filter language needs to be extended to include an "indirect push" operator, as well as arithmetic operators to assist in addressing-unit conversions.

The current filter mechanism deals with 16-bit values, requiring multiple filter instructions to load packet fields that are wider or narrower. It is possible that direct support for other field sizes would improve filter-evaluation efficiency. The existing read-batching mechanism clearly improves performance for bulk data transfer; a write-batching option (to send several packets in one system call) might also improve performance.

In addition to these problems, which may be regarded as deficiencies in the abstract interface, there is room for improvement in the existing implementation. During evaluation of each filter instruction, the interpreter verifies that the instruction is valid, that it doesn't

overflow or underflow the evaluation stack, and that it doesn't refer to a field outside the current packet. Since the filter language does not include branching instructions, all these tests can be performed ahead of time (except for indirect-push instructions); this might significantly speed filter evaluation. Even more speed could be gained by compiling filters into machine code, at the cost of greatly increased implementation complexity. Finally, with a redesigned filter language it might be possible to compile the set of active filters into a decision table, which should provide the best possible performance.

Idiosyncrasies of the 4.3BSD kernel create other inefficiencies. For example, because 4.3BSD network interface drivers strip the data-link layer header from incoming packets, the packet filter may be spending a significant amount of time to restore these headers. Also, in order to mark each packet with a unique timestamp, the packet filter calls a kernel subroutine called *microtime*; on a VAX-11/780, this costs about 70 uSec, probably more than the timestamp is worth.

8. Summary

The performance of the packet filter is clearly better than that of a user-level demultiplexer, and the performance of protocol code based on the packet filter is clearly worse than that of kernel-resident protocol code. Since the packet filter is just as flexible as a user-level demultiplexer, we believe that in systems where context-switching has a substantial cost, it is the right basis for implementing network code outside the kernel.

Are the advantages of user-level network code, even with the packet filter, worth the extra cost? Our experience has convinced us that in many cases, it is. The performance of such code is quite acceptable, and it greatly eases the task of developing protocols and their implementations. The packet filter appears to put just enough mechanism in the kernel to provide decent performance, while retaining the flexibility of a user-level demultiplexer.

Acknowledgments

Many people have used or worked on the packet filter implementation over the years; without their support and comments it would not be nearly as useful as it is. Especially notable are those who ported the code to other operating systems: Jon Reichbach of Ridge Computers, Inc., Glenn Skinner of Sun Microsystems, Inc., and Charles Hedrick of Rutgers University, who ported it to Pyramid Technology's system. Steve Deering and Ross Finlayson of Stanford made the VMTP measurements possible. We would like to thank the program committee and student reviewers for their comments.

References

1. Ed Basart. The Ridge Operating System: High performance through message-passing and virtual memory. Proceedings of the 1st International Conference on Computer Workstations, IEEE, November, 1985, pp. 134-143.
2. David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. "Pup: An internetwork architecture." *IEEE Transactions on Communications COM-28*, 4 (April 1980), 612-624.
3. David Boggs and Edward Taft. Private communication. 1987.
4. David R. Cheriton. "The V Kernel: A software base for distributed systems." *IEEE Software* 1, 2 (April 1984), 19-42.
5. David R. Cheriton. VMTP: A Transport Protocol for the Next Generation of Communication Systems. Proceedings of SIGCOMM '86 Symposium on Communications Architectures and Protocols, ACM SIGCOMM, Stowe, Vt., August, 1986, pp. 406-415.
6. David R. Cheriton and Willy Zwaenepoel. "Distributed process groups in the V kernel." *ACM Transactions on Computer Systems* 3, 2 (May 1985), 77-107.
7. Communications Machinery Corporation. *DRN-1700 LanScan Ethernet Monitor User's Guide*. 4th edition, Communications Machinery Corporation, Santa Barbara, California, 1986.
8. Computer Systems Research Group. *Unix Programmer's Reference Manual*, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. Computer Science Division, University of California at Berkeley, 1986.
9. *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 1.0)*. Digital Equipment Corporation, Intel, Xerox, 1980.
10. *TOPS-20 User's Guide*. Digital Equipment Corporation, Maynard, MA., 1980. Form No. AA-4179C-TM.
11. *LANalyzer EX 5000E Ethernet Network Analyzer User Manual*. Revision A edition, Excelan, Inc., San Jose, California, 1986.
12. Ross Finlayson, Timothy Mann, Jeffrey Mogul, Marvin Theimer. A Reverse Address Resolution Protocol. RFC 903, Network Information Center, SRI International, June, 1984.
13. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN, June, 1982, pp. 120-126.
14. Robert Gurwitz. Private communication. 1986.
15. ISO. ISO Transport Protocol Specification: ISO DP 8073. RFC 905, Network Information Center, SRI International, April, 1984.
16. M. Kirk McKusick, Mike Karels, and Sam Leffler. Performance Improvements and Functional Enhancements in 4.3BSD. Proc. Summer USENIX Conference, June, 1985, pp. 519-531.
17. Robert M. Metcalfe and David R. Boggs. "Ethernet: Distributed packet switching for local computer networks." *Communications of the ACM* 19, 7 (July 1976), 395-404.
18. *The Sniffer: Operation and Reference Manual*. Network General Corporation, Sunnyvale, California, 1986.
19. Jon Postel. Internet Protocol. RFC 791, Network Information Center, SRI International, September, 1981.
20. Jon Postel. Transmission Control Protocol. RFC 793, Network Information Center, SRI International, September, 1981.
21. D. M. Ritchie and K. Thompson. "The UNIX timesharing system." *The Bell System Technical Journal* 57, 6 (July/August 1978), 1905-1929.
22. Sun Microsystems, Inc. *Unix Interface Reference Manual*. Sun Microsystems, Inc., Mountain View, California, 1986. Revision A.
23. Brent B. Welch. The Sprite Remote Procedure Call System. UCB/CSD 86/302, Department of Electrical Engineering and Computer Science, University of California — Berkeley, June, 1986.

This Page Blank (uspto)

19



Europäisches Patentamt
European Patent Office
Office européen des brevets

11 Publication number:

**0 251 584
A2**

12

EUROPEAN PATENT APPLICATION

21 Application number: 87305416.7

51 Int. Cl.⁴: G06F 9/46

22 Date of filing: 18.06.87

30 Priority: 26.06.86 CA 512479

43 Date of publication of application:
07.01.88 Bulletin 88/01

64 Designated Contracting States:
AT DE FR GB IT NL SE

71 Applicant: **NORTHERN TELECOM LIMITED**
600 de la Gauchetiere Street West
Montreal Quebec H3B 4N7(CA)

72 Inventor: **Goyer, Pierre**
32 Terrasse David
Gatineau Quebec J8V 1G4(CA)
Inventor: **Selic, Branislav Vladeta**
7 Whelan Drive
Nepean Ontario K2J 2A3(CA)

74 Representative: **Crawford, Andrew Birkby et al**
A.A. THORNTON & CO. Northumberland
House 303-306 High Holborn
London WC1V 7LE(GB)

54 **A synchronization service for a distributed operating system or the like.**

57 A synchronization service which can be incorporated into a distributed operating system as a shared service. It allows the realization of different custom-built synchronization strategies for different applications. This approach is based on defining a general set of application-independent synchronization primitives. These are provided by the distributed operating system in the form of a synchronization service. By themselves the individual primitives are insufficient to provide synchronization. However, they can be combined in different ways to realize customized synchronization strategies. This leaves the ultimate responsibility for synchronization with the application, but in a much simplified form. Application programs can combine these primitives to construct the most suitable form of synchronization.

EP 0 251 584 A2

A SYNCHRONIZATION SERVICE FOR A DISTRIBUTED OPERATING SYSTEM OR THE LIKE

This invention relates generally to the field of computers, and more specifically to a synchronization service for use with computers.

Background of the Invention

As computing tasks increase in size and complexity, one approach to speed up the execution of these tasks is to use distributed programs. A distributed program can be defined as a computer program which is partitioned into multiple concurrent components which execute on separate processing sites which do not share a common memory.

In this context, the term "program" is used to imply a global objective (i.e. common goal). Each component (or portion) of the program performs some portion of the overall activity required to attain this common goal. Thus, a distributed program represents a set of (functionally) tightly-coupled components operating in a (physically) loosely-coupled environment.

Asynchronous operation of concurrent cooperating activities results in the time-dependencies and race conditions which can lead to errors. For example, two processes attempting to simultaneously update a shared variable may interfere with each other so that an incorrect value is assigned to the variable. The solution to such problems is through synchronization. Synchronization can be defined as the organization of actions and interactions of a system of concurrent asynchronous entities for the purpose of achieving some common objective.

One example of a distributed computer system is the case of a replicated database where each copy of the database is on a separate processing element. When a change is made to one copy then this change must be propagated to all the others if consistency is to be maintained. This involves synchronization. The situation may be complicated further if two or more conflicting changes are initiated simultaneously on different copies. In that case synchronization is required not only to ensure that all copies end up in the same state but also that the resulting state is valid. Other situations where synchronization is necessary include the restoration of the current state to new or recovering copies and the handling of failures.

Distributed synchronization can also be useful in standby schemes where redundant components are configured for greater system availability. In this case the components have to agree as to which will be the active and which the standby components and must also arrange for proper switchover in case of failures.

The cited examples illustrate the diverse ways in which synchronization is used in distributed systems. As can be expected, different applications can have different demands on synchronization: some may require fast response while others may place more emphasis on reliability and fault tolerance. This indicates that the choice of the most suitable synchronization technique and its implementation can only be made if the particular needs of the application are considered.

Unfortunately, in a large system supporting many different types of distributed application programs, leaving synchronization entirely to the application program could result in excessive duplication of effort, unreliable design, and suboptimal utilization of resources. Even worse, perhaps, is the possibility that the relatively complex issue of synchronization could dominate the design to such an extent that functional concerns are neglected. From that point of view a trusted, system-based, synchronization facility is preferred.

There are several important characteristics of distributed programs which make them significantly more difficult to design and implement compared to conventional non-distributed programs:

(1) Concurrent execution. This means that there is no single sequential control thread such as represented by the execution trace of a non-distributed program. Concurrency introduces timing dependencies among the system components which can lead to deadlocks or instability.

(2) Significant communication delays. The exchange of information between components of a distributed program involves non-negligible and randomly-distributed transmission delays. If these delays are comparable to the rate at which the components change state, the system may become unstable.

(3) Partial failure modes. Failures of distributed components require complex detection and recovery algorithms which are difficult to design and verify. Two types of partial failures exist:

- Communication path failures can result in the duplication, temporal reordering, or total loss of information being exchanged; and
- Processing component failures (hardware and software) lead to temporary loss of functionality.

The recovering action for each type of failure is quite different. Unfortunately, it is often difficult to distinguish them on the basis of the observed symptoms.

From the definition of synchronization it can be seen that the need for synchronization is determined by the shared objective of the cooperating distributed entities. This common objective places interdependencies on the individual entities so that a change in the state of one necessitates appropriate changes (reactions) in others. This can be expressed as a requirement to preserve certain application-dependent state consistency constraints. The problem of maintaining consistency is further complicated by the fact that each entity, in addition to internal interactions, is also exposed to independent interactions with the environment. (The environment consists of other distributed components which do not share the same objective as the synchronized system, but which use it for their own purposes). This means that the stimulus to change state can occur simultaneously in two or more synchronized entities. The synchronization problem can then be viewed as one of ordering concurrent interdependent activities.

The simplest form of ordering which guarantees consistency is serialization: the execution of activities one at a time. Although synchronization strategies exist which are not based on serialization, they will not be considered here due to their relative complexity.

Two basic, and not necessarily exclusive, classes of strategies exist for achieving serialization in distributed systems;

(1) Centralized strategies.

In this case, the ordering of activities is performed by a unique distinguished entity. Synchronized entities, with externally induced work requests, first approach the distinguished entity for permission. This entity resolves concurrent requests by granting a right to only one of the competing entities. When that entity completes its work, the right is granted to another entity, and so on.

A major feature of this type of scheme is that there is a single point of control. This allows the implementation of relatively complex yet reliable and efficient scheduling algorithms. Examples of centralized strategies can be found in A Decentralized Control Method in a Distributed System by J.P. Cabanel et al, Proceedings 1st Conference, Distributed Proc. Systems, Huntsville, AL, 1979 and

in A Failure Tolerant Centralized Mutual Exclusion Algorithm by G. N. Buckley et al, Proceedings 4th Conference, Distributed Computer Systems, San Francisco, Ca. 1984.

(2) Distributed strategies.

In this case, there is no central scheduler. Instead, ordering is accomplished through distributed agreement. Key to this scheme is a shared "clock" (logical or physical). This is generally a monotonically increasing numeric variable which is maintained consistently by all the synchronized entities. Work requests are timestamped with the clock value at the time of arrival and then processed in order. However, because two or more requests can be concurrent (i.e., they have the same timestamps), ties are resolved through group negotiation: a new work request is first broadcast to all other entities which respond either with a simple acknowledgement or a work request of their own. Once an entity is aware of all concurrent work requests within the group, it orders them according to some tie-breaking rule and then processes them. Since each entity uses the same ordering algorithm each will perceive the same sequence of events as all the others.

The distinguishing feature of distributed strategies is that operation does not depend on a single critical entity at any time. This makes them very fault-tolerant. However, they are generally less efficient than centralized strategies when the number of entities to be synchronized is large. Examples of distributed strategies can be found in Time, Clocks, and the Ordering of Events in a Distributed System by L. Lamport, Comm. ACM, (21,7), July 1978, in An Algorithm for Maintaining the Consistency of Multiple Copies by D. Herman et al, Proceedings, 1st Conference Distributed Proc. Systems, Huntsville, AL., 1979 and in Synchronization in Distributed Programs by F.B. Schneider, ACM Transactions on Prog. Lang. & Syst, (4,2), April, 1982.

Combinations of these two forms, such as the circulating sequencer proposed in Algorithms for Distributed Data Sharing Systems Which Use Tickets by G. Le Lann, Proc. 3rd Berkeley Workshop on Dist. Data, Aug. 1978, are possible. In that scheme, a centralized controller is used to control the clock used for timestamping. (Although the controller function is circulated among the distributed entities, at any given time it is performed by only one entity.) The ordering of activities is then done in a distributed fashion, based on timestamp values and a tie-breaking rule.

The following patents depict examples of distributed processing in general, and attention is directed to them: U.S. patent 3,411,139 dated November 12, 1968 by J.T. Lynch et al; U.S. patent 3,631,405 dated December 28, 1971 by G.S. Hoff et al; U.S. patent 3,771,137 dated November 6, 1973 by R.P. Barner et al; and U.S. patent 4,115,866 dated September 19, 1978 by J.L.G. Janssens et al.

Summary of the Invention

One objective of the Synchronization Service of the present invention is to provide a set of application-independent capabilities which would allow the construction of specific synchronization strategies belonging to the categories listed above. To do this it must incorporate the essential abstract features of those strategies. These are defined in the form of a general synchronization paradigm described in a following section.

Because of concurrent execution and the possibility of partial failures, it is necessary to closely synchronize the operation of the distributed components of a program. Synchronization can be defined as the ordering of actions and interactions of components in a distributed program so that the state of each component remains consistent with the common goal.

Experience with concurrent systems has shown that the synchronization problem is difficult to solve even for non-distributed situations; the number of possible component interactions is usually very large, increasing the probability of a design error.

A further difficulty is caused by the fact that no single synchronization strategy is adequate for all distributed programs. If multiple distributed programs are to be supported on a system, this means that the synchronization problem may have to be solved in many different ways.

Given the diversity of synchronization strategies and the difficulty of implementing them, is it possible to provide some assistance to designers of distributed programs to increase the reliability of their designs?

The approach to this problem, presented by the present invention, consists of providing a set of primitive synchronization operators at the level of a distributed operating system. Such operators can be used to construct more complex forms of synchronization customized to different applications. This approach has the following advantages:

- It provides a one-time trusted implementation of common mechanisms;
- It does not favour any particular synchronization strategy which would favour some applications but

penalize others;

-It provides a systematic framework (programming model) for designing and implementing distributed programs.

The operating system component which implements the synchronization primitives (operators) is called the Synchronization Service.

The essential idea behind the Synchronization Service is that the synchronization problem can be tackled hierarchically. Each level in the hierarchy may have different synchronization mechanisms based on the synchronization facilities of the levels below. The lower levels of this hierarchy can be designed to be application-independent and can therefore be provided as a reliable system service. This, in turn, increases the reliability of programs and reduces development time.

This approach to distributed synchronization attempts to decompose the synchronization problem. At the lowest level of decomposition a general set of application-independent synchronization primitives is defined. These are provided by the distributed operating system in the form of a synchronization service 10. By themselves the primitives are insufficient to provide synchronization. However, they can be combined in different ways to realize customized synchronization strategies. This leaves the ultimate responsibility for synchronization with the application program, but in a much simplified form. The role of the synchronization service 10 is to hide many of the more basic housekeeping functions inherent in distributed synchronization. For instance, all fault-tolerant synchronization schemes require a monitoring function to keep track of the operational status of all relevant distributed components. The present invention consolidates such a function as a system service where it can be shared by many application programs.

Stated in other terms, the present invention is a general service, provided within a distributed operating system, which can be used by application and system programs to implement synchronization between program components that are physically distributed.

Stated in other terms, the present invention is a synchronization service for use with a computer having a distributed operation system, to allow the construction of a customized synchronization scheme, for synchronizing the constituent portions of a distributed program, the service comprising: a general set of application-independent synchronization primitives, whereby the construction of the customized synchronization scheme is achieved by the selective implementation of the application-independent synchronization primitives.

Stated in yet other terms, the present invention is a synchronization service for use with a computer having an operating system distributed over a plurality of processing elements, to allow the construction of a customized synchronization scheme, for synchronizing the constituent components of a distributed program, the service comprising the steps of:

a) joining a program component on a first processing element to a group of existing program components on at least a second processing element so that each of the existing components is aware of the present and location of the joining components;

b) informing each member of the group of physically distributed program components when one or more components which are members of the group, depart from it;

c) selecting, as a distinguished member, one program component from a group of distributed program components such that, within the group, there is never more than one distinguished member; and

d) providing mutually exclusive rights to the group of distributed program components such that no more than one component can appropriate a given right at any time.

Stated in still other terms the present invention is a synchronization service, for use with a computer having an operating system distributed over a plurality of processing elements, to allow the construction of customized synchronization schemes for synchronizing the constituent components of a distributed program, the service including a synchronization master control comprising: master control means for activating the synchronization service; polling means for polling the processing elements associated with the components of the distributed program so as to monitor the status of the processing elements; control means for joining new members to the group, and for handling departures of members from the group; and a database means containing information representative of the current state of the synchronization service at a given point in time.

Brief Description of the Drawings

The present invention will now be described in more detail with reference to the accompany drawings, wherein like parts in each of the several figures are identified by the same reference character, and wherein:

Figure 1 depicts a simplified block diagram of the synchronization service of the present invention;

Figure 2a is similar to Figure 1 but is for one specific embodiment thereof;

Figure 2b is a variation on the embodiment of Figure 2a;

Figure 2c is similar to Figure 2b;

Figure 3a is a chart depicting the primitives and corresponding replies employed by the invention;

Figure 3b is a symbolic representation of the constituent tasks of synchronization master control of Figure 1;

Figure 3c is a symbolic representation of the constituent tasks of member agent 11 of Figure 1;

Figure 4 is a simplified functional flow diagram for a database;

Figure 5 is a simplified functional flow diagram for a database;

Figures 6 to 8, 9a, 9b, and 10 to 13 inclusive represent action sequences helpful for understanding the operation of the present invention; and

Figure 14 is a simplified representation of the usage dependencies helpful in understanding the operation of the present invention.

Detailed Description

Synchronization service 10 is based on a general distributed program paradigm. This paradigm is represented by the concept of synchronization groups. A synchronization group is a set of distributed program components called "members", and referred to by the reference character 18, which cooperate to achieve a common objective. Note that members 18 are not a part of synchronization service 10, but they use synchronization service 10.

In other words, the distributed operating system 15, to which synchronization service 10 is applied, will support both distributed application programs and distributed system programs. Both the distributed application and system programs consist of several program components (called members 18) which in turn consist of subcomponents called tasks. In synchronization service 10 there is one synchronization group for each distributed application or system program.

A primitive synchronization operator has effect only within the domain of a particular synchronization group. Synchronization groups, therefore, encapsulate units of tightly coupled distributed functionality. Of course, synchronization service 10 allows many synchronization group to coexist on a single distributed operating system 15.

The basic construct of synchronization service 10 is the synchronization group representing a set (i.e., a system) of distributed entities which are tightly coupled to each other in some way. The

state and action dependencies which bind these entities are not specified at this level so that synchronization groups are decoupled from application semantics.

Formally, a synchronization group is a set of components, called members 18, in which each group ideally has the following properties:

(1) Uniqueness: There can be any number of synchronization groups in a larger system but each synchronization group is distinguished from all others by a unique synchronization group identifier.

(2) Physical distribution: Each member of a synchronization group exists on a different processing element 12. (This is simply a matter of convenience: extending the concept of synchronization groups to logically distributed entities is possible). Note that there are no restrictions concerning the number of synchronization groups which may have members 18 on a particular processing element 12. This means that two or more synchronization groups can overlap in physical space.

(3) Reliable communication: Communication between any pair of members 18 is non-lossy, non-duplicating, and order-preserving. Furthermore, full connectivity is assumed; i.e., each member 18 can communicate directly with all other members 18. If the physical system does not have these properties then it is assumed that an underlying communication service exists which provides them. The intent here is to isolate communications issues from synchronization issues.

(4) Dynamic behavior: Members 18 can depart or join the synchronization group at any time and independently of each other. (The group exists as long as at least one member 18 exists.) Departures may be either application-driven or due to processing element 12 failure. This property captures the dynamic nature of real-world components.

(5) Mutual exclusion: Each synchronization group maintains a set of shared objects called rights, each of which can be either free or associated with at most one member 18. They are functionally equivalent to semaphores (reference: E.W. Dijkstra, Cooperating Sequential Processes, Technical Report EWD-123, Technological University, Eindhoven, 1965) but for a distributed environment. (However, a member 18 can hold more than one right at a time.) A departing member 18 cannot abscond with a right since any rights it holds are automatically freed. In essence, rights are a general mechanism for distinguishing between group members 18. The assignment of functional significance to rights is up to the application.

(6) Distinguished member: One and only one member 18 of every synchronization group is designated as its distinguished member. The appointment is made at random and is transferred to

another member 18 if the current distinguished member 18 departs. This property is intended to serve those synchronization strategies which require a central coordinator although synchronization service 10 makes no assumptions regarding the functional significance of the distinguished member 18. (Note that the distinguished member feature is simply a special case of the mutual exclusion property but has been singled out purely for convenience.) Since the selection and preservation of a distinguished member 18 is by synchronization service 10, application programs need not implement their own election algorithms.

A synchronization group represents a unit of synchronization. The facilities of the synchronization service 10 (described later) are all limited in scope to the respective synchronization group.

The synchronization problem is often formulated as a problem of maintaining data consistency in a dynamic environment. From that point of view, the synchronization service 10 ensures consistency of the following information sent to members 18:

- (1) current membership list;
- (2) the identity of the distinguished member;

and

- (3) the status of all group rights.

This information is maintained consistently and correctly in the face of continual departures and arrivals of members 18.

The concept of synchronization groups does not encompass application program-level consistency; that is the responsibility of the application program. Instead, a synchronization group maintains a consistent view (on all its members 18) concerning the status of its objects: the list of active members 18, the status of rights, the distinguished member designation. These responsibilities are therefore removed from the view of the application program.

Figure 1 depicts a simplified block diagram of synchronization service 10 of the present invention. A distributed application program, structured as a synchronization group, typically has members 18 (i.e. distributed program components) which are physically distributed across two or more processing elements 12a...12n (referred to collectively as processing elements 12). In the implementation of Figure 1, the structure of synchronization service 10 matches the structure of the synchronization group by providing a local synchronization controller, i.e. member agent 11, for each group member 18. Thus, there is a separate implementation of synchronization service 10 for each application program; note, however, that there is only one synchronization master control 13 regardless of how

many implementations, and only one synchronization agent 14 per processing element 12. Each implementation is functionally independent of the others.

Note that the group of processing elements 12 together with processing element 19 form part of the distributed computing environment (i.e. distributed operating system 15) which synchronization service 10 is designed to synchronize. Not also that each processing element 12 may have a plurality of member agents 11, and that processing element 19 may be combined with one of the processing elements 12.

Member agents 11 provide the main interface to the synchronization service 10. Application program components (i.e. members 18) initiate synchronization activities by invoking the desired synchronization primitives (to be described later). This is communicated to the local member agent 11 which then interacts with other member agents 11 in order to effect the specified synchronization function. The member agent 11 also informs the members 18 of synchronization requests initiated by other members 18 as well as group events such as the failure of active members 18 and the joining of new ones.

Member agents 11 are dynamic entities which follow the dynamics of the application programs they serve. A member agent 11 is created (by the local synchronization agent 14) when an application program component (i.e. member 18) requests to be synchronized with other members 18 in a synchronization group. It is destroyed when the member 18 is unsynchronized.

To ensure coherent behaviour of synchronization service 10, control of the individual implementations of the service 10 is centralized. This is done through a three-level hierarchy with a unique master controller at the top (i.e. synchronization master control 13), an intermediate layer of controllers in the middle (i.e. synchronization agents 14a to 14n, referred to collectively as synchronization agents 14), and a layer of member agents 11 at the bottom. This hierarchy allows a decomposition of the control problem into smaller more comprehensive subproblems. Note from Figure 1 that there is one synchronization agent 14 for each processing element 12, and it controls all the member agents 11 in that processing element 12.

Figure 2a is similar to Figure 1, but depicts a specific embodiment of the synchronization service, referred to by reference character 100 as applied to distributed operating system 115. In Figure 2a there is a synchronization master control 13 on processing element 119, three processing elements 112a, 112b, and 112c, three synchronization agents 14a, 14b, and 14c, along with six members 18a to 18f along with their corresponding

member agents 11a to 11f respectively. In the distributed computing example of Figure 2a, processing elements 112a, 112b, 112c and 119 are each an IBM PC-AT. Note that the members 18a to 18f inclusive are not part of synchronization service 100 while everything else shown in Figure 2a is. Members 18a to 18f inclusive use the synchronization service 100. Note also that there is another synchronization master control (not shown) on standby.

Figure 2b is similar to Figure 2a, but is further simplified and depicts only those items that constitute one implementation of synchronization service 100 (i.e. implementation 100a). That is, members 18a and 18e (Figures 2a and 2b) form one synchronized group. Members 18b, 18c, 18d, and 18f (Figure 2a) form at least one other synchronized group.

Figure 2c is a simplified application to exemplify synchronization service 100a of Figure 2b. In Figure 2c the hardware implementing synchronization service 100a is a group of IBM personal computers of the AT series, linked by an IBM LAN (local area network) 226. That is, processing element 112a is an IBM PC-AT computer 212a, processing element 112b is an IBM PC-AT computer 212b, and processing element 119 is an IBM PC-AT computer 219.

In Figure 2c, computer 212a is a telephone operator's workstation as is computer 212b. The application in Figure 2c is to maintain a telephone directory and to allow the user at both computers 212a and 212b to have access to the telephone directory, to access it to determine an individual's telephone number, and to be able to update the telephone directory as changes occur. Computer 219, in this example, handles the tasks of synchronization master control 13 and database 16 (Fig. 2b).

Returning now to the general case of Figure 1, the role of synchronization master control 13 is to provide internal synchronization between the components of the local synchronization service 10. In essence, it performs those functions where a consistent (but not necessarily correct) view of the system 15 is required. More precisely, synchronization master control 13 is responsible for:

(1) Activation of synchronization service 10.

This is done by activating the synchronization agents 14 as the processing elements 12 are restarted.

(2) Monitoring of processing elements 12.

This function involves observing (polling) the status of all processing elements 12 by communicating with local synchronization agents 14. Any changes in these states are detected by synchronization master control 13 and appropriate notifications are dispatched to the synchronization service components affected by the change.

(3) Management of synchronization groups.

synchronization master control 13 is the central arbiter for all synchronization groups in the local synchronization service 10. It is involved in handling transient conditions which occur in group operation:

- group establishment,
- joining of new members 18, and
- departures of joined members 18.

Note that synchronization master control 13 does not participate in the steady-state operation of synchronization groups and, consequently, is not normally a performance bottleneck.

Synchronization master control 13 must be highly fault-tolerant since synchronization service 10 may be used to implement standby schemes by applications. For that reason it is backed up by at least one other instance operating in standby mode. If the currently active synchronization master control 13 fails, the standby will take its place. Because this is the Synchronization Service, the selection of an active synchronization master control 13 from the set of instances must be done through an internal agreement (election). This is the only place in the entire system where the synchronization service 10 cannot be used for such a purpose. However, in this case, the problem occurs in a very specific context and can be solved in a specific way (for example, by using a bully algorithm for a distributed election as described in Elections in a Distributed Computing System by H. Garcia-Molina, IEEE Trans. on Computers, (C-31,1), Jan. 1982).

Once the active synchronization master control 13 has been selected, the standby resorts to a monitoring mode in which it periodically polls the active instance until a failure is detected.

Since a standby is used, following a failure of the synchronization master control 13, its previous state must be reconstructed on the standby, preferably without involving the application program. This can be achieved through the information kept by the synchronization agents 14. As a consequence,

except for slightly extended service times due to the recovering process, application programs are unaware of synchronization master control 13 failure.

SYNC MASTER TASK

The Sync Master Task 20 is the root task (i.e. program) of the synchronization service 10 control hierarchy. It provides the central control point for all synchronization groups. It consists of four main subcomponents as depicted in Figure 3b and is located within synchronization master control 13. The four main subcomponents of the Sync Master Task 20 are as follows:

SYNC MASTER CONTROL 21 establishes and maintains the operational state of the Sync Master Task 20. This includes the Sync Master recovery algorithm. Sync Master Control 21 consists of the main procedure of the Sync Master Task 20.

POLLING CONTROL 22 is responsible for detecting failure of processing elements 12. This subcomponent sends periodic messages to all synchronization agents 14. If a reply is not received within a certain time interval (after several retries have been attempted) the corresponding processing element 12 is declared as failed and a notification is sent to all remaining synchronization agents 14. This subcomponent is implemented within the Sync Master Task 20.

SYNC AGENT CONTROL 23 deals with events which occur at the processing element 12 level. This subcomponent is responsible for activating newly-recovered synchronization agents 14 as well as for accepting notifications, from the synchronization agents 14, about the arrivals and departures of group members 18. These are then relayed to the appropriate Group Control 24. This subcomponent is also implemented within the Sync Master Task 20.

GROUP CONTROL 24 handles events which are relevant to one group. This includes the joining and departure of group members 18. The Group Control function is implemented by the Group Master Task 25. There is one such task 25 for each synchronization group. Tasks 25 are created dynamically by the Sync Master Task 20.

The tasks comprising the Sync Master Task 20 maintain a shared database 16 (Figure 1) which represents a snapshot of the current state of the synchronization service 10. This database is described later.

SYNCHRONIZATION AGENT

A synchronization agent 14 resides in the control program of each processing element 12 which requires synchronization service 10 and it is the sole representative of the Sync Master Task 20 in that processing element 12. The synchronization agent 14 has the following responsibilities:

- It accepts SYNCHRONIZE directives and creates corresponding member agents 11.

- It monitors the status of all active member agents 11 on its processing element 12 and detects their disappearance (spontaneous or planned).

- It notifies the synchronization master control 13 of all changes (arrivals and departures) of Member Agents 11 on its processing element 12.

The synchronization agent 14 is implemented by the Sync Agent Task which is part of the operating system 15 on the corresponding processing element 12.

The synchronization agents 14 are permanent representatives of synchronization master control 13 within their host processing element 12. They have three main purposes:

(1) Synchronization agents 14 are a focal point for controlling all member agents 11 within a single processing element 12. This reduces the load on synchronization master control 13 which simply sends common control information to synchronization agents 14 for distribution to local member agents 11.

(2) Synchronization agents 14 isolate member agents 11 from the effects of synchronization master control 13 failures. All communication between the synchronization master control 13 and Member Agents 11 is channeled through the synchronization agents 14. If the synchronization master control 13 is temporarily unavailable (due to failure), the synchronization agents 14 will hold member agent 11 messages destined for the synchronization master control 13 until the latter is reinstated. In this way failures of the synchronization master control 13 are masked from member agents 11 and hence the applications.

(3) Synchronization agents 14 participate in the recovery of the synchronization master control 13. When a synchronization master control 13 is being reinstated it can reconstruct its operational state simply by querying all the synchronization agents 14. This is much faster and more reliable than querying the member agents 11 since these are more dynamic and more numerous.

The synchronization master control 13 maintains a database 16 (Figure 1) which represents the current state of the synchronization service 10 within the system 15. The database can be accessed through two keys:

- by group identifier --for access to the data for a

particular synchronization group, and
-by processing element identifier --for access to synchronization service components located on a particular processing element 12.

The basic structure used is the linked list of dynamically allocated control blocks, each block corresponding to some synchronization service component. This represents a trade-off between the requirement to minimize storage costs and the need for fast access to the data.

The next section describes the operation of the internal mechanisms used to achieve the synchronization functions. In the following discussion the communication between member agents 11 is assumed to be reliable; i.e., it is non-lossy, non duplicating, and order preserving. If the communication medium is unreliable an underlying reliable communication service provided within the distributed operating system can be used.

Rights are a set of shared objects within each synchronization group; each right can be free or associated with at most one member 18. One example of a right is a database lock whereby only one user at a time can write to a database and no one else can read or write at that time. See also the "Update" right referred to later.

Rights are distributed in a centralized fashion since that minimizes overhead and complexity. In principle, this can be done by any member agent 11. For convenience, the control and distribution of rights are performed by the distinguished member (one of the members 18). The distinguished member 18 already has the uniqueness and fault-tolerant properties which are also required by the controller for rights. Thus, the Member 18 selected as the distinguished member has to perform this special function in addition to its standard synchronization functions. The selection of a distinguished member is done, by the synchronization master control 13, at the time the group is established (see below).

When a member 18 requires a right, its member agent 11 directs the request to the distinguished member 18. If the right is available, the distinguished member 18 will grant the right and inform the requesting member agent 11. If the right is already appropriated, then depending on the type of request made, the request is either queued by the distinguished member 18 or it is refused. In the first case, requests are handled on a first-come first-served basis.

Should the current distinguished member 18 fail, a new one is appointed by the synchronization master control 13 (which is also responsible for detecting the failure). Of course, until a new distinguished member 18 is appointed, rights cannot be distributed or retrieved, but all the other synchronization services are still available. In order to mini-

mize the effect of a distinguished member 18 failure, the state of rights is reconstructed to the point just prior to failure. Each member 18 keeps a list of all rights which it has appropriated as well as a list of all its outstanding rights requests. This information is then exchanged with the new distinguished member 18 which can then assume the same state as the previous distinguished member 18. The entire switchover process is transparent to the application program.

If a member 18 fails, the distinguished member will automatically release any rights held by that member 18 and also purge any queued requests generated by that member 18.

Member agent 11 is the main functional component of synchronization service 10 and is responsible for handling all directives initiated by the user. It performs four classes of functions as depicted in Figure 3c and as represented by the following:

The COMMUNICATIONS HANDLER 33 provides a reliable (order-preserving, non-lossy, non-duplicating) communications service between group members 18; in order to minimize deadlocks the communication mode used is asynchronous message passing. This function is required only if there is no reliable communications service present within the distributed operating system 15.

The GROUP STATE HANDLER 32 maintains a local version of the current state of all the other group members 18.

The DIRECTIVE HANDLER 31 provides the interface between user tasks (components of members 18) and the member agent 11.

The DM HANDLER 30 implements the distinguished member functionality and is active on only one member 18 of the group at a time. This member 18 is selected by the Group Master Task 25 (Figure 3b). The distinguished member 18 is responsible for allocation of rights as well as for broadcasting group status change notifications to all other members 18 of the group. (This information is received from the Group Master Task 25.)

Member agents 11 are created dynamically by the synchronization agent 14 in response to a SYNCHRONIZE directive (Primitive). They are also destroyed by the synchronization agent 14 after they have left the group or following a failure.

Broadcasts and Acknowledgements

When an application program initiates a broadcast (via the GROUP-BROADCAST primitive), its local member agent 11 distributes the information to all other active member agents 11. It then accumulates acknowledgements until all active member agents 11 have replied after which the application program is notified (via the GRP-ACK reply signal).

If an element 12 fails before its acknowledgement is dispatched, the broadcasting member agent 11 will assume an implicit acknowledgement from that member so that failures will not disrupt the application.

Group Establishment and Joining of New Members

A newly joining member 18 first informs the synchronization master control 13 (via its synchronization agent 14) of its intent to join the synchronization group. The synchronization master control 13 then determines if this is the first reported member of the group. If it is, then this Member 18 is designated as the distinguished member 18 and a notification is sent back. This establishes the group.

If the group is already established, synchronization master control 13 registers the new member 18 as being in the joining state and informs the group's distinguished member agent 11. Upon receiving this notification the distinguished member agent 11 broadcasts a join request to all member agents 11 on the list and waits for the corresponding group acknowledgement. The period between the broadcast of the join request and the full acknowledgement of that request by all joined member agents 11 is called the joining interval. During that time some member agents 11 will become aware of the new member agent 11 before others. This opens up the possibility that some messages broadcast within the group may bypass the partially synchronized member agent 11. If messages received by this member agent 11 are passed to the application program, then the application program function of this member 18 would not necessarily perceive the same sequence of group events as other members 18; it could miss some. Therefore, the new member agent 11 must acknowledge any messages received from other member agents 11 (in order to satisfy the acknowledgement requirement) but, once acknowledged, the messages are discarded; i.e. they are not passed on to the application (an exception is messages containing other joining or departure requests which are processed by the member agent 11 but still not relayed to the application). This mode of operation remains in effect until the join

request is finally acknowledged by the entire group. At that point, the new member 18 informs its application that it is fully joined and switches to normal operation. The overall effect, as perceived by the application, is that the joining operation is atomic.

The handling of messages that were discarded during the joining interval is no different to the application program than the handling of messages missed by the member 18 while it was down; that is, once synchronized with the group, the application program must proceed to upgrade its functional state to be consistent with the functional states of other members 18. The best method for achieving this depends on the application program.

Departure of Members

The departure of a member 18 from a synchronization group occurs when the member 18 decides to unsynchronize or when the host processing element 12 fails. In the former case, the departing procedure is as follows: the synchronization group (i.e. agent 11) notifies the Sync Master Task 20 of its intention. This event is relayed, via the appropriate Group Master Task 25 (Figure 3b), to the distinguished member 18 of the group. The distinguished member 18 then broadcasts this information to all other group members 18. Note that there is one Group Master Task 25 for every synchronization group defined in service 10.

In the case of a processing element 12 failure, the failure is detected by the Polling Control 22 within Sync Master Task 20 (Figure 3b) and the same sequence as described above is executed.

If the departed member 18 was a distinguished member, Group Master Task 25 will first select a new distinguished member 18 and then proceed in the same manner as above.

The synchronization agents 14 are intermediaries between synchronization master control 13 and the member agents 11. Synchronization agents 14 are created and dispatched when their host processing element 12 is initialized. Upon creation they wait to be contacted by the synchronization master 13, if one exists. Any application level requests for synchronization are queued until an acknowledgement is received from synchronization master control 13.

During normal operation, the synchronization agents 14 serve as a relay point for communication between the synchronization master control 13 and the member agents 11. All communication is buffered until acknowledged by the receiver so that the Member Agents 11 are protected from tem-

porary failures of synchronization master control 13. The synchronization agents 14 also extract and store any information relevant to the reestablishment of the synchronization master control 13.

Most of the operation of the synchronization master control 13 has already been described above. The only aspect remaining is the monitoring function.

The monitoring of the existence of processing elements 12 is done by the Polling Control 22 which polls each individual synchronization agent 14. The failure of a processing element 12 implies that the corresponding synchronization agent 14 is down as well as all member agents 11 that were present on that processing element 12. When that happens the synchronization master control 13 notifies all affected Group Master Tasks 25. These, in turn, inform their distinguished member agents 11 which then broadcast this information to other member agents 11.

Before we go any further, it may be advantageous to introduce the primitives used with synchronization service 10. The primitives can be split into two categories:

(1) Synchronous Primitives are in the form of request-reply pairs; member agents 11 submit requests for some action to be performed on their behalf and synchronization service 10 eventually matches these with appropriate replies.

(2) Asynchronous Notifications are spontaneous signals informing a member agent 11 about changes in the status of its group or conveying a message sent by some other member agent 11.

There are only two types of asynchronous notifications that can be sent to a member agent 11:

-GROUP-CHANGE (group status) is sent when a new member 18 has joined or an active member 18 has departed from the group. The status information includes the complete new membership list and the id of the new distinguished member.

-GROUP-MSG (message) signals the arrival of a message from some other member 18 (broadcast or point-to-point).

The application program must allow forms of communication (i.e. synchronous and asynchronous) although it may choose to handle asynchronous communications in a synchronous manner by ignoring them until the current activity sequence is complete.

The synchronous primitives and corresponding replies are depicted in chart form in Figure 3a, to which attention is directed.

The primitives are:

-SYNCHRONIZE (group-id)

This is a directive which is issued by a member 18 (via its member agent 11) when it wishes to become synchronized with the group specified by <group-id>. If no group exists at the time, one is established. The only signal expected in reply to this directive is the SYNCH-DONE signal.

-SYNC-DONE (group-status)

This is a signal from the synchronization service 10 (i.e. member agent 11) in response to a successful synchronization of a member 18 following the invocation of the SYNCHRONIZE directive. The return parameter, <group status>, contains the same information about the status of the group as the GROUP-CHANGE primitive described below. It includes a <dm-flag> parameter which informs the member 18 if it is the bearer of the distinguished member status.

- UNSYNC

This directive is used when a member 18 decides to depart from its group. It ensures orderly deactivation.

-UNSYNC-DONE

This signal is a confirmation that the member 18 has been removed from its synchronization group.

-GROUP-CHANGE (group-status)

This is an asynchronous signal which is generated by the member agent 11 whenever a new member 18 joins the group or when a member 18 departs from the group. If this member 18 is the new distinguished member as a result of the change, a <dm-flag> parameter in the <group-status> data record will be set appropriately. The treatment of this situation is left to the application program. The new status of the group is also returned.

-REQ-RIGHT (right-id, mode)

This directive is issued when a member 18 needs exclusive access to a group right. If the right is available, it is guaranteed to be granted to only one requesting member 18 (there may be multiple

simultaneous requests for the same right). If the right is not available, then if the <mode> parameter specifies a "queued" request, it is queued until it can be serviced. Alternatively, if the <mode> parameter specifies "immediate" the request is refused since the right has already been appropriated by another member 18 of the group.

10 - R-GRANTED (right-id)

This signal informs a member 18 that it has been granted the required right.

15 -R-REFUSED (right-id)

This signal informs a member 18 which has requested a right, with the "immediate reply" mode specified in the request, that the right is not available. (If a queued request was made then this signal will never be generated.)

25 -REL-RIGHT (right-id)

This directive is used to release an appropriated right.

30 -R-RELEASED (right-id)

This signal is the reply to the REL-RIGHT directive.

35 -QRY-RIGHTS

This is a directive which is used to obtain a snapshot of the distribution of group rights among group members.

40 -R-STATUS (rights-status)

This is a reply signal to the QRY-RIGHTS directive. The <rights-status> parameter lists, for each group right, the member-id of the member which owns it, if any.

Note that service 10 cannot guarantee the currency of the returned information since changes in the distribution of rights can occur at any time.

-GRP-BRDCST (message)

This directive is used to broadcast a synchronization event (message) to all synchronized members 18. It is the responsibility of the synchronization service 10 (via member agent 11) to ensure that all members 18 receive the message. The <message> parameter can be used to timestamp the synchronization event. The higher level software is responsible for supplying this parameter as well as interpreting its functional significance.

-GROUP-ACK

This is an acknowledgement signal for the GRP-BRDCST directive. It signifies that all members 11 have received the latest broadcast message.

-SND-TO-MEM (message)

This directive is used to send a point-to-point message to another group member 18.

-MSG-ACK

This is an acknowledgement that the latest point-to-point message has been received by the destination member 18.

Before the invention is described further, it may be of value to give some brief examples of the application of the primitives.

The first example is the control of a standby configuration. In this configuration there are two or more distributed program components (i.e. members 18) each on a different processing element 12, each of which is equally capable of providing the necessary function. Only one should be active at any given time while the others are standing by, ready to be activated should the active one fail. Assuming that they are all part of the same synchronization group that the algorithm which each member 18 executes is the same (the synchronization service primitives are highlighted in capitals):

```

SYNCHRONIZE;
Wait for SYNC-DONE signal;
If not selected as the distinguished member then
Repeat
Listen for SYNC-CHANGE signals;
until selected as the distinguished member;
Execute function;

```

If a member 18 is not selected as the distinguished member following synchronization with the group, then it simply waits until it is designated as the distinguished member.

The next example concerns the updating of a replicated database, i.e. the same example mentioned in the Background of the Invention. In this case there are multiple instances of a database, each of which can initiate an update request as a result of external activity. Such requests will be called external to distinguish them from "shadow requests". Shadow request are copies of an external request which a member 18 sends to all other members 18 so that they can make the appropriate changes to their copies of the database. For brevity, the handling of any other requests except update requests is ignored.

The solution shown below uses the mutual exclusion feature of the synchronization group. A right, called the Update right, is defined. The holder of this right is the member 18 whose request will be honored; all other members 18 must withhold their requests and perform the shadow request sent by the holder of the right:

Solution A

```

Repeat
Wait for next request;
If external request then
begin
REQ-RIGHT (update);
While waiting for R-GRANTED
Handle any incoming shadow requests;
GRP-BRDCST (external request);
Handle external request;
REL-RIGHT (Update);
end
else
Handle shadow request;
until termination;
Note that the application program need not be
concerned with spontaneous failures of other mem-
bers 18 since that is handled by the synchroniza-
tion service 10.

```

An important problem which must be handled by this application (i.e. Solution A, above) is the addition of new or recovering instances. These will not necessarily have the same state as the others and therefore must be brought to the same functional level. The situation is complicated by the possibility that updates may be initiated at other instances while the new instance is being upgraded. One method of dealing with this is for the new instance to appropriate the Update right to ensure that the state remains unchanged while it is being upgraded. The algorithm performed by a restarting instance is then:

```

SYNCHRONIZE;
Wait for SYNC-DONE signal;
REQ-RIGHT (Update);

```

While waiting for R-GRANTED.
 Discard any shadow requests received;
 Obtain current copy of database;
 REL-RIGHT (Update);

Following this, the normal request processing algorithm described above (i.e. Solution A) is executed.

The current copy of the database is obtained from any other member 18 through an internal protocol using point-to-point messages (i.e. SND-TO-MEM directives). Instead of a copy of the entire database it may be more convenient to request an update log and then perform the updates missed while the member 18 instance was down.

Figure 4 is a functional flow diagram representing the synchronization service 10 database 16 when accessed through the processing element 12 identifier.

The head and tail pointers (AGT-LST-HD and AGT-LST-TL) respectively, point to a linked list of synchronization agent control blocks (tAGT-CB) for those synchronization agents 14 involved.

There is one synchronization agent control block tAGT-CB for each processing element 12 which requires synchronization service 10. It contains a link (AGT-LST-LNK) to other synchronization agent control blocks tAGT-CB. This chain enables quick scanning of affected processing elements 12 when an entire block of processing elements 12 fails. Each synchronization agent control block tAGT-CB also contains a pointer (MMCB-LST-HD) to a chain of member agent control blocks (tMEM-CB) which reside on that processing element 12. Through this chain it is possible to detect quickly all synchronization groups which are affected by the failure of a processing element 12. Whereas all other chains in the synchronization service database 16 remain unchanged once they are established, this chain follows the dynamics of member 18 joinings and departures.

In order to simplify searching and list maintenance, the last Sync Agent control block tAGT-CB in the list is a dummy block.

Each member agent control block tMEM-CB corresponds to one member 18 of one synchronization group. Among other data, this control block contains a pointer (not shown in the diagram) to the corresponding synchronization agent control block tAGT-CB. This link allows quick reconfiguring of the processing element-Member Agent chain when necessary.

Figure 5 is a functional flow diagram representing the synchronization service database 16 when accessed through the unique group identifier.

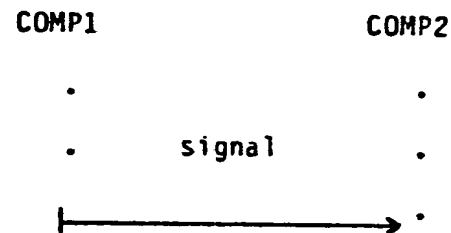
GRP-HDR [group id] is a static array of pointers. Each item of the array points to a circular list of member agent control blocks (tMEM-CB) of which belong to the same synchronization group.

The member agent control blocks tMEM-CB are linked into a circular list to facilitate selection of a distinguished member. This list grows as members 18 are added to the synchronization group, each successive block identified by the next available positive integer (MEM-ID). This integer corresponds to the member 18 identifier.

ACTION SEQUENCES

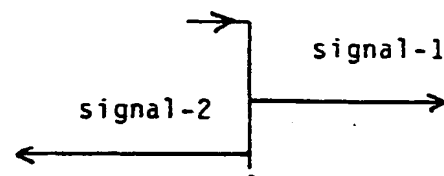
This section describes various action sequences within synchronization service 10. A diagrammatic representation is used to show these sequences. The following conventions are used.

- A full horizontal line indicates a message or rendezvous between two components (i.e. programs or tasks):



where signal is the name of the entry procedure in component COMP2 which accepts the signal. COMP1 is the component which sent the signal.

- a vertical line (1) following the reception of a signal indicates processing within the appropriate component which received the signal. This processing results in one or more signals being dispatched to other components:



- If an asterisk (*) appears next to a signal it implies that the signal may be repeated (to different destinations).

- A signal which is enclosed in braces (e.g., <signal>) indicates that the signal is not mandatory and may be omitted depending on circumstances.

• Bracketed numbers in the Figures (e.g. (1)) designate explanatory notes which contain textual descriptions pertaining to various signals. The notes are given in the text relating to the relevant Figure. As it is believed that the Figures are self-explanatory, only brief comments will be made regarding the figures.

Joining of New Members

An "empty" synchronization group is one in which no members 18 are active. When the first member 18 joins, it is designated as the distinguished member by default. Figure 6 depicts a member 18 joining an empty group. The sequence of events is as depicted in Figure 6, to which attention is directed. The abbreviations used in the Figures are as follows: APPL means an application task; SNYC AGT means the synchronization agent task; MEM AGT means the member agent task; SYNC MST means the synchronization master task 20; GRP MST means the Group Master Task 25; MEM AGT (DM) means the distinguished member agent task; APPL (DM) means the application task which corresponds to the distinguished member agent.

The following notes refer to the bracketed numbers in Figure 6.

Notes:

(1) The Sync Agent will create a member agent task only if it had not existed previously, otherwise it will REINIT a previously created task instance. The START-AGT signal which follows initialization is used to pass initial data to the member agent 11.

(2) The AGT-MST-MSG signal to the sync master 13 contains the complete information about all member agents 11 on this processing element 12, including the newly-created member 18. (This ensures state convergence even in the presence of design faults.) The MST-REPLY message is used for positive acknowledgement so that the Sync Agent can send the next message to the Sync Master if it has one. (Only one outstanding message is allowed between the Sync Master and a Sync Agent.)

(3) If this group has not previously existed, a new Group Master is created. In that case a STARTUP signal follows to pass initial data to the Group Master Task, and an ACTIVATE signal is used to force it into an operational mode. If the group had existed previously (but had lost all its members) the existing Group Master Task is used.

(4) Once the Group Master has been activated, a GRP-EVENT signal is sent by the Sync Master informing it of the joining of the first member.

(5) Upon receipt of the GRP-EVENT signal, the Group Master selects the newly-created member agent 11 as the distinguished member and sends it a GRP-STATE signal. This signal establishes a connection between the Group Master and the Distinguished Member. All subsequent GRP-STATE signals are sequenced to ensure proper event ordering as well as to guard against communication failures. The GRP-REPLY signal is used to acknowledge one or more GRP-STATE signals and provides reliable communication.

The GRP-STATE signal contains the complete new state of the group rather than just information about the changes. This ensures that the system will converge to the true state even in the presence of design faults.

(6) The CHECK-FAIL signal is used to poll the application task to detect unexpected failures of the application. The application task never receives this signal; however, should the task fail, the member agent task will be notified by the underlying operating system kernel.

(7) The SYNC-REPLY signal contains a reply code of SYNC-DONE.

Figure 7 depicts the sequence for joining an existing synchronization group; the Group Master Task already exists, and the distinguished member is used to notify (broadcast) the other members 18 of the presence of a new member 18.

Notes:

(8) If the application has so requested, a GROUP-CHANGE signal is sent by each member agent 11 to the application whenever it detects a change in status of the group. In the case of the distinguished member the status change is gleaned from the GRP-STATE signal.

(9) When the distinguished member receives a GRP-STATE signal which indicates a group change (not all do), it broadcasts the new state to all other group members 18 using a MEM-MSG signal. Each member 18 acknowledges such messages with an ACK signal to provide reliable communication.

(10) When the newly-joining member 18 receives its MEM-MSG from the distinguished member it will send a SYNC-REPLY signal to the application (instead of a GROUP-CHANGE signal).

The control flow for the departure of a member 18 is shown in Figure 8. Note that the case of a processing element 12 failure is not shown here but is instead treated separately.

Notes:

(11) The sequence shown here corresponds to a voluntary departure initiated by the application program issuing an UNSYNC directive. This results in the member agent task terminating which, in turn, sends a COMPLETE signal to the parent task, the Sync Agent. The sequence is similar in situ-

ations where the departure is not voluntary:

-When the application task fails, the member agent 11 is notified (through the failure of the CHECK-FAIL message) which results in the termination of the member agent task (and consequently, raising of the COMPLETE signal).

-If the member agent task itself fails, the Sync Agent is notified by the operating system kernel with a COMPLETE signal.

(12) Indicates a <GROUP-CHANGE> signal to an application task not shown in the Figure.

Figures 9a and 9b together depict the recovery of the synchronization master control 13 (i.e. Sync Master). The most general case is considered, i.e. the case of a running synchronization service 10 with active synchronization groups. This includes, as a subset, the case of a "cold" start.

Notes:

(13) Upon activation, Sync Master sends an ACTIVATE signal to each configured Sync Agent. The Sync Agent, whether they are already active or not, will respond with an AGT-REPLY message which includes a list of all member agents supported on that processing element. (The MST-REPLY signal is used for acknowledgement: refer to Figure 6.)

(14) Following activation of the Sync Agent, the Polling Control subcomponent 22 of Sync Master Control sends a POLL-AGT message to which the Sync Agent responds with an AGT-REPLY message. This exchange is repeated periodically to detect outages of the processing element.

(15) A Group Master is initiated only the first time a group is encountered. Refer to Figure 6 for further details on Group Master initiation.

(16) Before activating a Group Master, it is provided with data regarding the status of its members through GRP-DATA signals. Each signal contains the information for one member agent of one group. (This info is obtained from the AGT-REPLY messages.) In this way, the Group Master reconstructs the status of its group.

(17) After all Sync Agents have responded, the reconstruction is complete and an ACTIVATE signal is sent to all Group Masters. The Group Masters respond by sending a GRP-STATE signal to all distinguished members. Since this signal contains the complete group state, any group changes that might have occurred while the Sync Master was down are detected.

Figure 10 depicts the procedure for handling the failure of a processing element 12 which contains a synchronization agent 14 (Sync Agent).

Notes:

(18) The Sync Master detects a failure of a processing element when a TIME_OUT event is received. This means that a Sync Agent has not responded to a poll.

(19) For each group affected by the processing element failure, the Sync Master will send a GRP-EVENT signal to the respective Group Master.

5 Figure 11 depicts the procedure for the recovery of a processing element 12 which is part of synchronization service 10. Note that the recovery of processing element 12 does not extend to recovering member agent 11 tasks. It is assumed that these will be recovered when the application tasks (i.e. programs) which use them are restarted. Thus, the only action to recover a process element 12 is to integrate the sync agent 14 with the rest of synchronization service 10.

10 Notes:

(20) When a previously failed Sync Agent finally responds to a POLL_AGT signal, the Sync Master initiates the recovery procedure.

(21) The Sync Master registers the new processing element 12 and sends an ACTIVATE signal to the Sync Agent on that processing element 12 (Refer to Figure 9a for a more detailed description of the activation sequence).

Figure 12 depicts the procedure for member 18 to member 18 messages. This procedure (protocol) is used both for group broadcasts and point-to-point messages between members 18.

Notes:

(22) In case of a broadcast, a copy of the message is sent to each member 18. If the member 18 is not active, the message is not sent.

(23) Upon receiving a MEM-MSG signal which indicates an application-level message the message is relayed to the application task responsible for receiving asynchronous messages.

(24) A SYNC-REPLY (codes: GRP-ACK or MSG_ACK) signal is sent back to the originator. Figure 1 depicts the procedure employed in the processing of all directives which require distinguished member intervention (rights handling directives).

Notes:

(25) If the request is made on the distinguished member site, then no message is sent.

(26) A reply signal (MEM-MSG followed by a SYNC-REPLY to the application).

In one implementation made by the inventors, the code for synchronization service 10 was contained in ten files which were distributed into six units, the usage dependencies (and compilation order) of which are shown in Figure 14.

Notes:

50 SYNCCTRL contains the stub of the Sync Master unit and directives to include three files (SYNCMST, SYNCGMST, and SYNCPLL) which implement the Sync Master function. It also contains the definitions required for the master database 16. SYNCMST is an "include" file which

contains the code for the Sync Master Task.

SYNCGMST is an "include" file which contains the code for the Group Master Task.

SYNCPOLL contains the Polling Control 22.

SYNCLOCL contains the stub of the Sync Agent unit as well as a definition of data and procedure objects shared by the Sync Agent Task and the Member Agent Tasks. It also contains directives to include two files (SYNCAGT, SYNCMAGT).

SYNCRESX contains the definition of the SYNCHRONIZE primitive. This unit must be loaded with the application code which uses the synchronization service.

SYNCDEFI contains a set of internal compile-object definitions for the synchronization service. This includes the definition of all entries used for communication between synchronization service components which are hidden from user programs. Since this file contains only compile objects it is not loaded.

SYNCDEFX contains a definition of all compile-objects which are exported by the synchronization service to its users. Since this file contains only compile objects it is not loaded.

Application tasks which use the synchronization service need to include SYNCRESX and SYNCDEFX in their usage lists.

Note that SYNCCTRI implements the function of synchronization master control 13 (Fig. 1), and that SYNCLOCL and SYNCRESX together implement the functions of member agent 11 and synchronization agent 14 (Fig. 1). The files SYNCDEFI and SYNCDEFX are not resident in the service; they can be thought of as tools used in the construction of the synchronization service but they are not themselves a part of it.

Simplified pseudocode listings for the main constituents of the invention follow as appendix I. They are believed to be self-explanatory. Any elaboration of the material is accomplished through the use of appended notes, to which attention is directed.

As a further aid to the understanding and to the use of the present invention the following (a copy of a "User's Reference" to the synchronization service of the present invention, as prepared by one of the inventors) is included as Appendix II. It will expand on the use of the present invention.

Claims

1. A synchronization service [10] for use with a computer having a distributed operating system, to allow the construction of a customized synchronization scheme, for synchronizing the constituent portions of a distributed program, said service comprising:

a general set of application-independent synchronization primitives, whereby the construction of said customized synchronization scheme is achieved by the selective implementation of said application-independent synchronization primitives.

2. The synchronization service of claim 1 wherein said application-independent primitives comprise the following functions: synchronize; synchronize done; and unsynchronize.

3. The synchronization service of claim 2 wherein said primitives further comprise the following functions: request right; right granted; right refused; release right; group broadcast; and group acknowledge.

4. The synchronization service of claim 3 wherein said primitives further comprise the following functions: unsynchronize done; send to member; and message acknowledge.

5. A synchronization service [10] for use with a computer having an operating system [15] distributed over a plurality of processing elements [12], to allow the construction of a customized synchronization scheme, for synchronizing the constituent portions [18] of a distributed program, said service comprising:

a common synchronization master control means [13];

a synchronization agent means [14] for each processing element;

a plurality of application program components [18], each component located on a different processing element, each said component having associated therewith a member agent [11], said member agent being a program for interfacing with said synchronization agent means, and said synchronization agent means interfacing between said master control means and said member agent, whereby a customized synchronization scheme can be constructed based upon a general set of application-independent synchronization primitives contained in both said synchronization agent means [14] and said member agent [11] and accessed via said synchronization agent means.

6. The synchronization service of claim 5 wherein said application-independent primitives comprise the following functions: synchronize; synchronize done; and unsynchronize.

7. The synchronization service of claim 6 wherein said application-independent primitives further comprise the following functions: request right; right granted; right refused; release right; group broadcast; and group acknowledge.

8. A synchronization service [10] for use with a computer having an operating system [15] distributed over a plurality of processing elements, to allow the construction of a customized synchroniza-

tion scheme, for synchronizing the constituent component of a distributed program, said service [10] comprising the steps of:

a) joining a program component [18] on a first processing element [12] to a group of existing program components [18] on at least a second processing element [12] so that each of the existing components is aware of the presence and location of the joining components;

b) informing each member of a group of physically distributed program components when one or more components which are members of said group, depart from it;

c) selecting, as a distinguished member, one program component from a group of distributed program components such that, within said group, there is never more than one said distinguished member; and

d) providing mutually exclusive rights to said group of distributed program components such that no more than one said component can appropriate a given right at any time.

9. The synchronization service of claim 8 further including the step of providing reliable point-to-point communication between said distributed program components on the basis of their internal group identifiers.

10. The synchronization service of claim 9 further including the step of providing a broadcast mechanism from any one program component to all other program components which are currently declared as being in the same group as the broadcasting component.

11. The synchronization service of claim 10 wherein said program components are components of an application program.

12. The synchronization service of claim 10 wherein said program components are components of an operating system program.

13. The synchronization service of claim 8 wherein said physical processing elements are logically distributed entities at one physical location.

14. A synchronization service [10], for use with a computer having an operating system [15] distributed over a plurality of processing elements [12], to allow the construction of customized synchronization schemes for synchronizing the constituent components [18] of a distributed program, said service comprising, as required, the steps of:

a) establishing a synchronization group for said distributed program, said group comprising at least one distributed program component [18];

b) joining a program component [18] to said group of existing program components so that each of the components is aware of the presence and the location of all the other components in said group;

c) informing each member of said group of distributed program components when one or more components which are members of said group, depart from it;

d) selecting, as a distinguished member for said group, one program component from said group of distributed program components such that, within said group, there is never more than one said distinguished member; and

e) providing mutually exclusive rights to said group of distributed program components such that no more than one said component can appropriate a given right at any time.

15. The synchronization service of claim 14 further including the step of providing full connectivity between all said distributed program components of said group.

16. The synchronization service of claim 15 wherein said distributed program is an application program.

17. The synchronization service of claim 15 wherein said distributed program is an operating system program.

18. The synchronization service of claim 15 wherein each said program component [18] is on a different processing element [12].

19. A synchronization service [10], for use with a computer having an operating system [15] distributed over a plurality of processing elements [12], to allow the construction of customized synchronization schemes for synchronizing the constituent components [18] of a distributed program, said service including a synchronization master control [13] comprising:

master control means [21] for activating said synchronization service;

polling means [22] for polling the processing elements [12] associated with said components of said distributed program so as to monitor the status of said processing element;

control means [24] for joining new members [18] to said group, and for handling departures of members [18] from said group; and

a database means [16] containing information representative of the current state of said synchronization service at a given point in time.

20. The synchronization service of claim 19 further including, at each said processing element, a synchronization agent [14] comprising:

means for accepting synchronization directives and for creating corresponding member agents; and

means for monitoring the status of all active member agents on said processing element and reporting same to said synchronization master control [13].

21. The synchronization service of claim 20 further including at each said processing element [12], a member agent [11] each synchronization

group, comprising:

communications means [33] for providing a reliable communications service between program components [18];

storage means [32] for maintaining a local version of the current state of all other program components [18];

handler means [31] for providing the interface between user tasks and said member agent [11]; and distinguished member means [30] for implementing the distinguished member function on only one program component [18] at any given time.

5

10

15

20

25

30

35

40

45

50

55

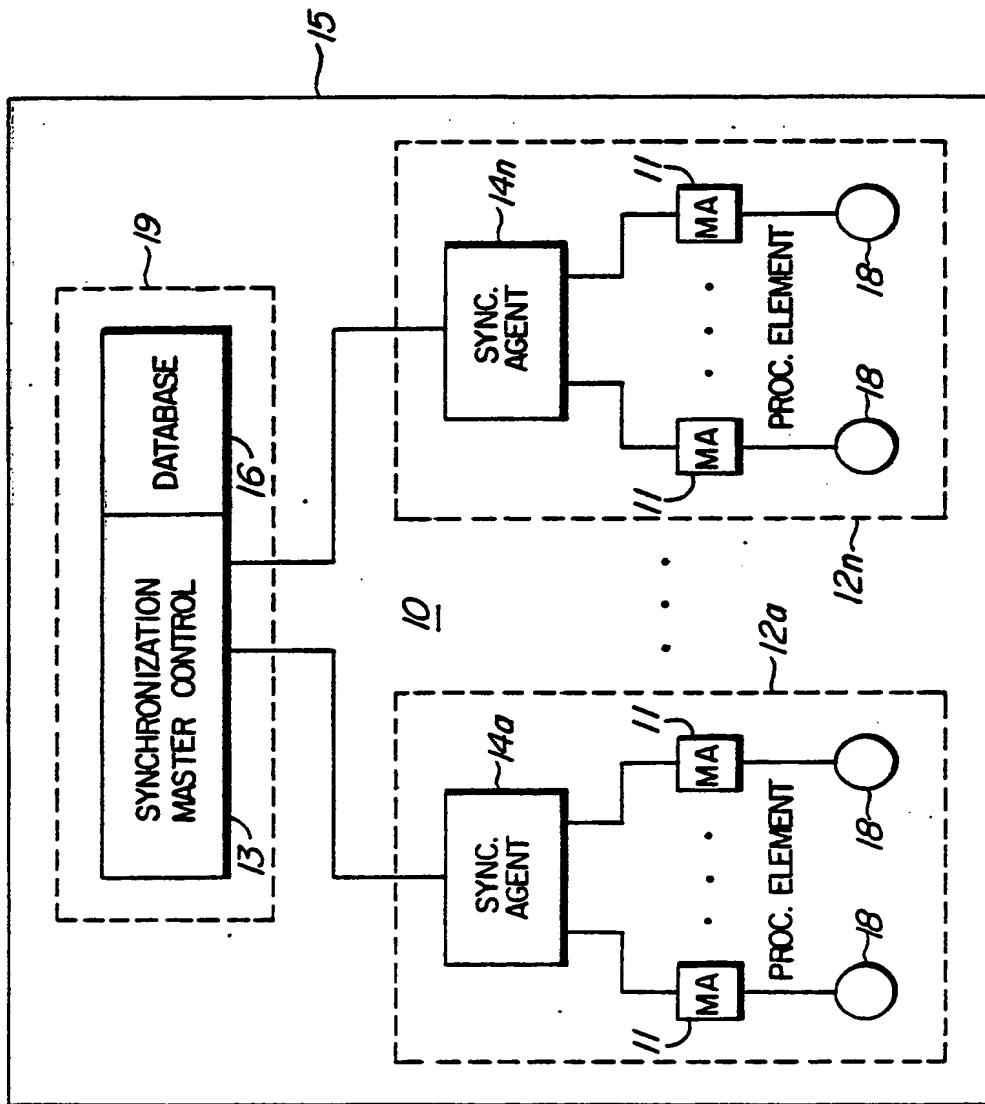


FIG. 1

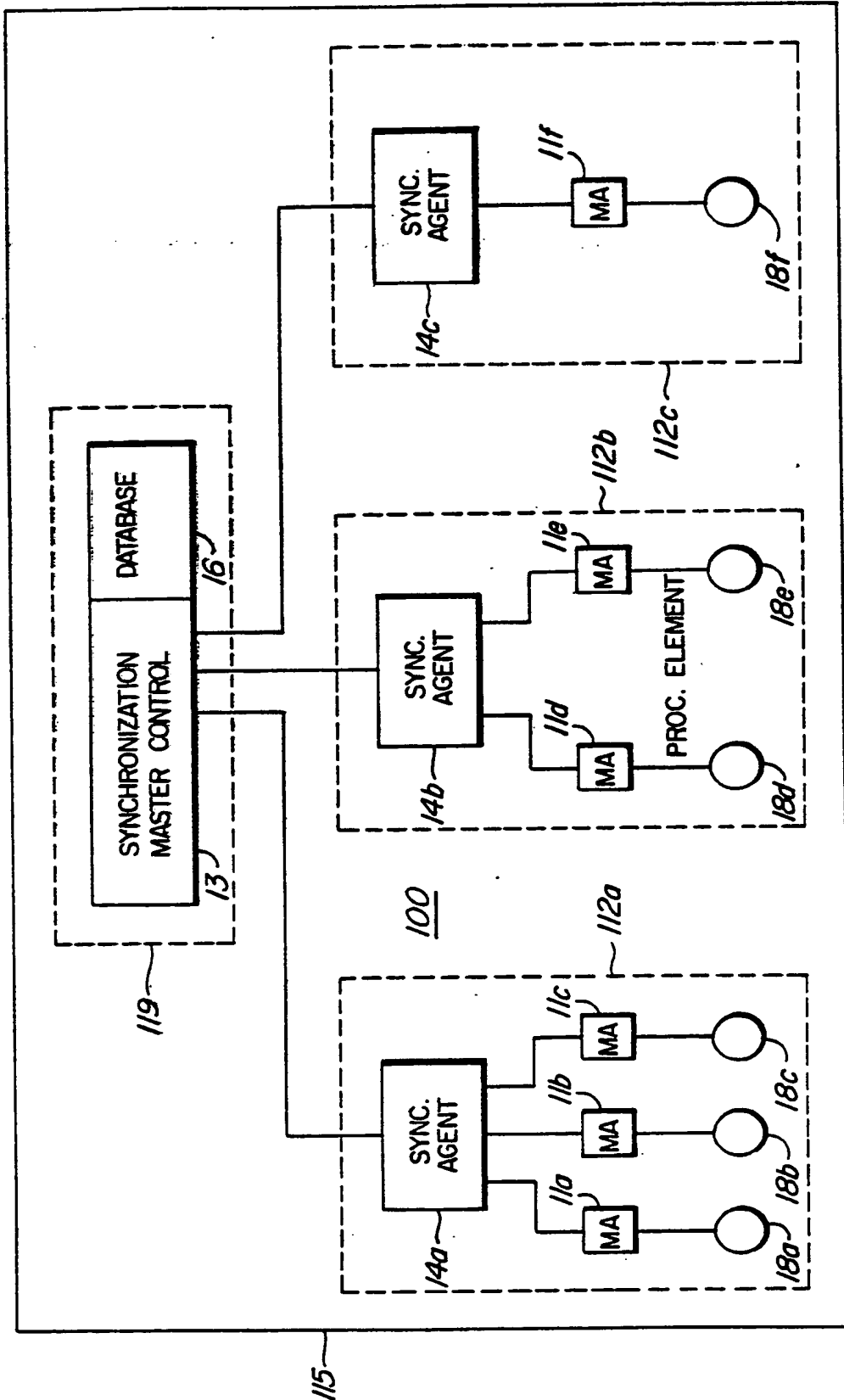


FIG. 2a

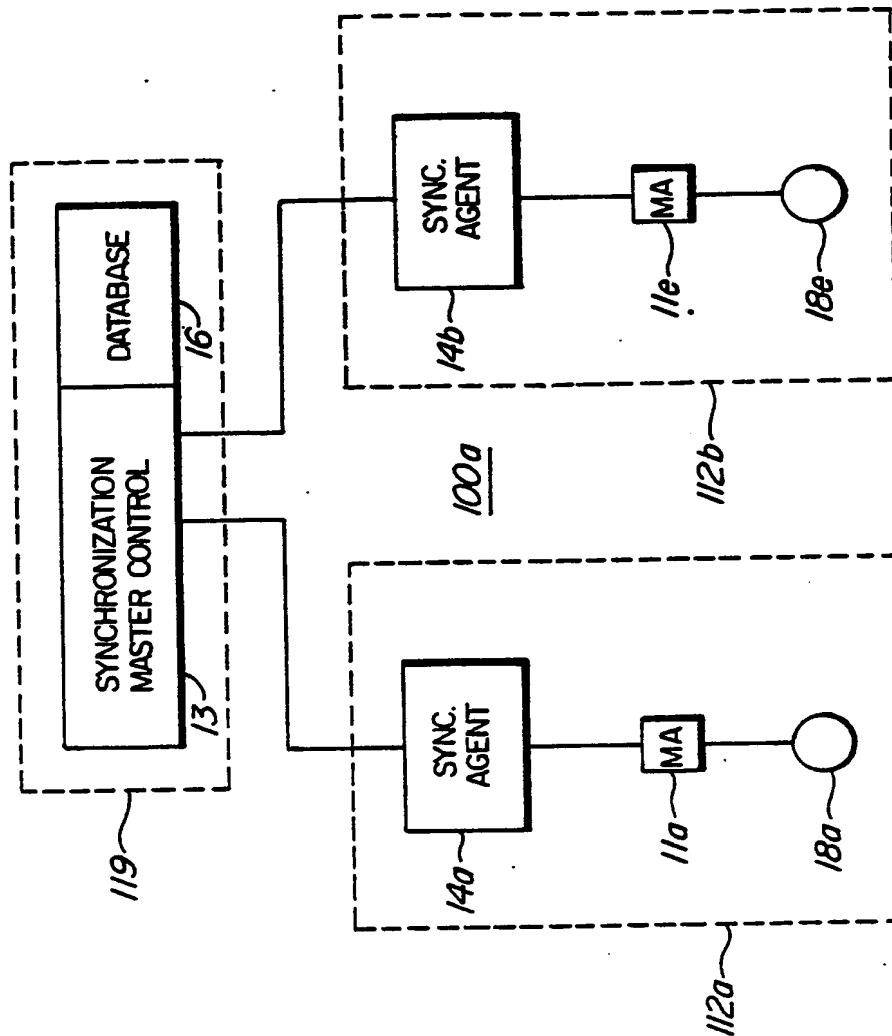


FIG. 2b

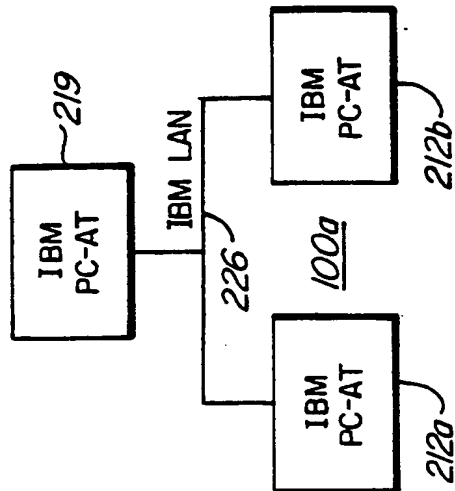


FIG. 2c

SYNCHRONOUS PRIMITIVES

REQUEST	REPLY
SYNCHRONIZE (group-id)	SYNC_DONE (group-status)
UNSYNC	UNSYNC_DONE
REQ_RIGHT (right-id, Immed)	R_GRANTED
	R_REFUSED
REQ_RIGHT (right-id, Queued)	R_GRANTED
REL_RIGHT(right-id)	R_RELEASED
QRY_RIGHTS	R_STATUS (rights-status)
GRP_BRDCST (message)	GRP_ACK
SND_TO_MEM (message)	MSG_ACK

ASYNCHRONOUS NOTIFICATIONS

GROUP-CHANGE (group-status)
GROUP-MSG (message)

FIG. 3a

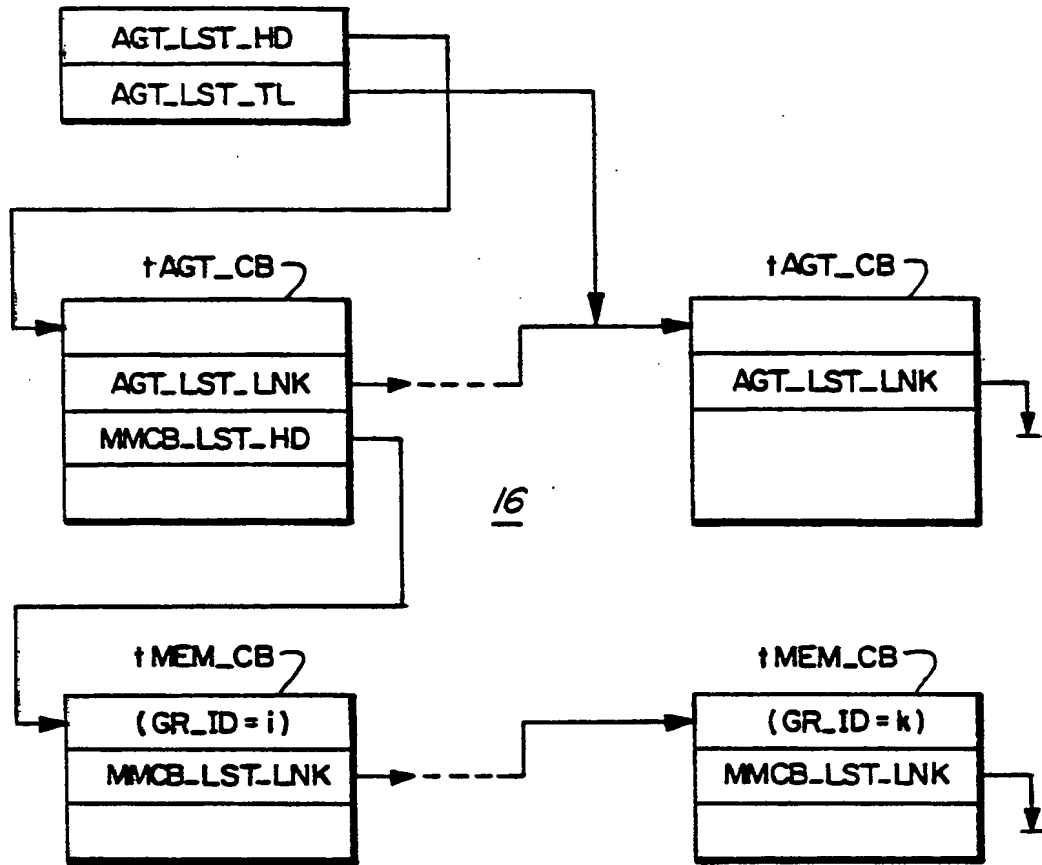


FIG. 4

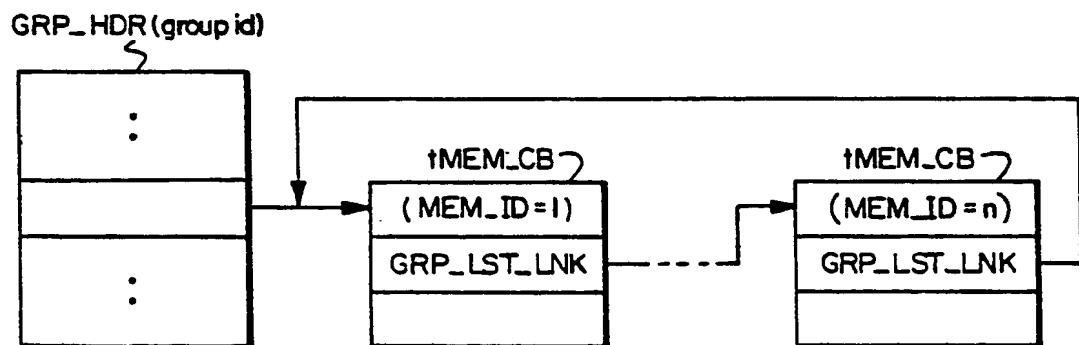


FIG. 5

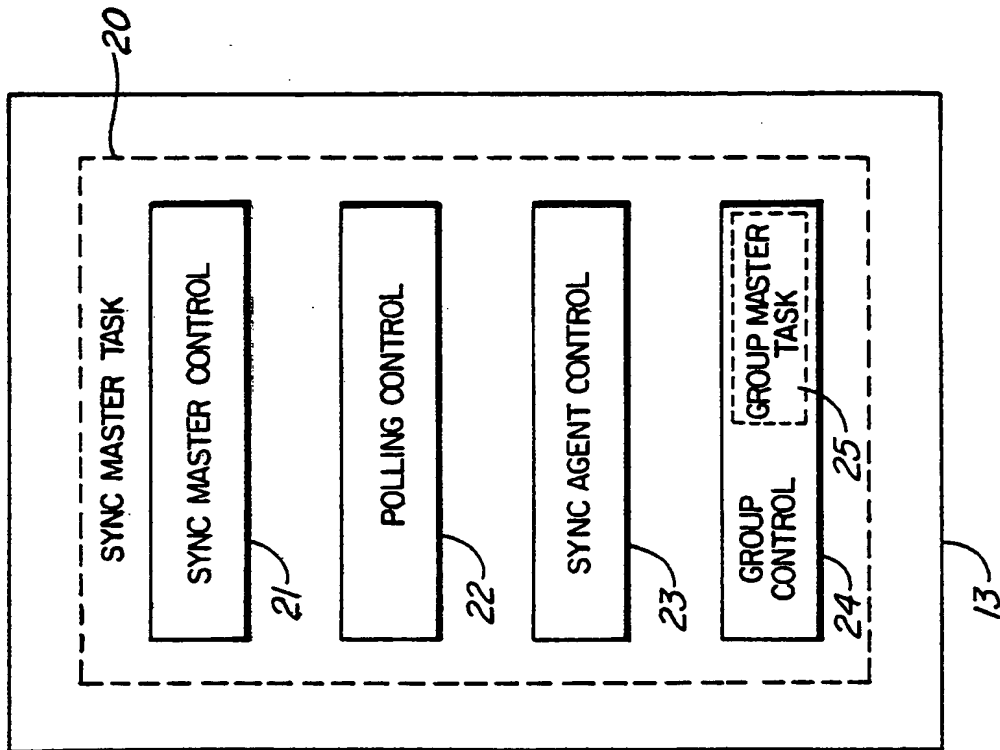


FIG. 3b

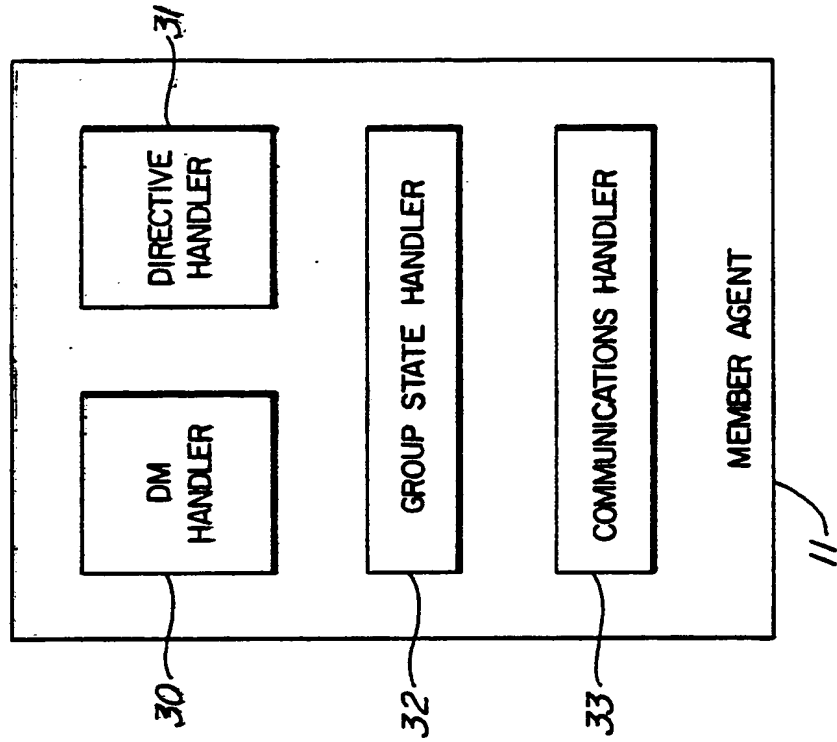
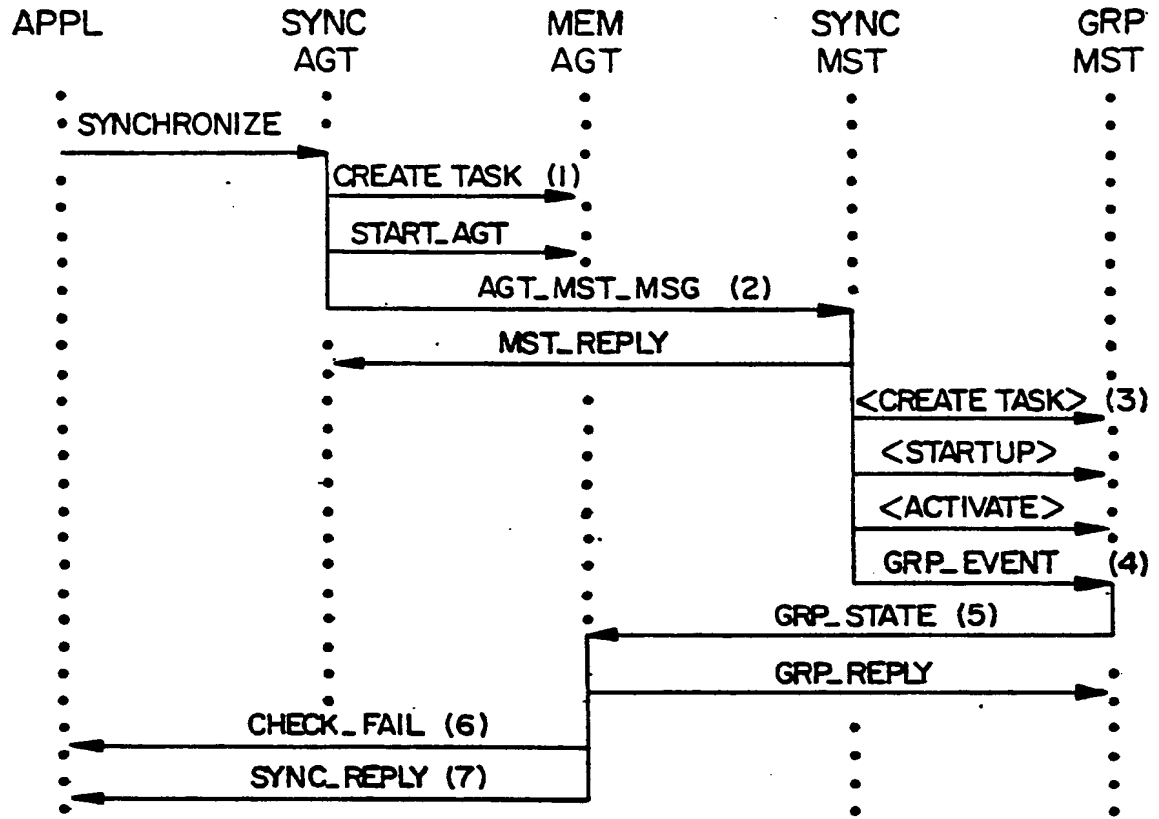


FIG. 3c



LEGEND
APPL = MEMBER 18
SYNC AGT = SYNCHRONIZATION AGENT 14
MEM AGT = MEMBER AGENT 11
SYNC MST = SYNCHRONIZATION MASTER 13
GRP MST = GROUP MASTER 25

FIG. 6

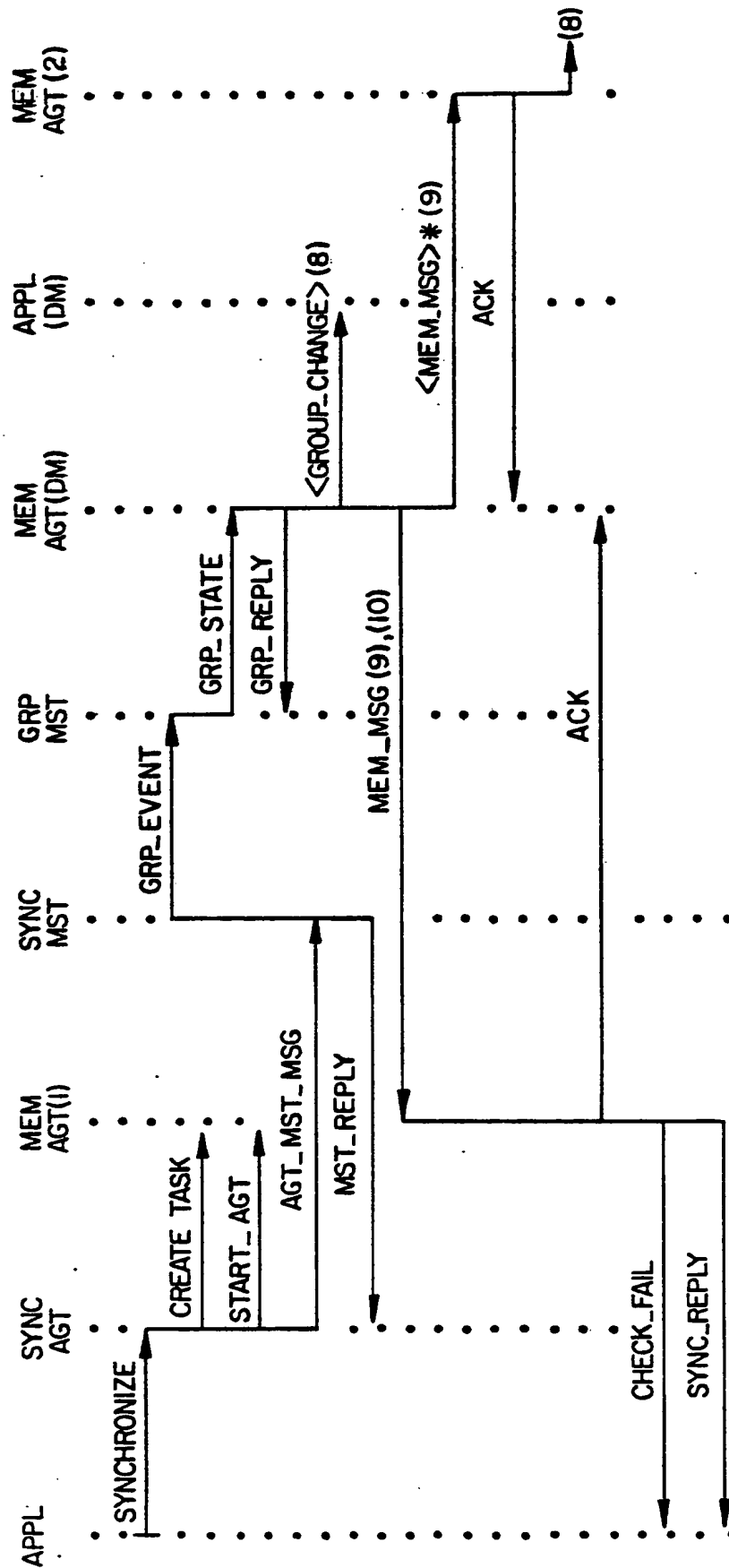


FIG. 7

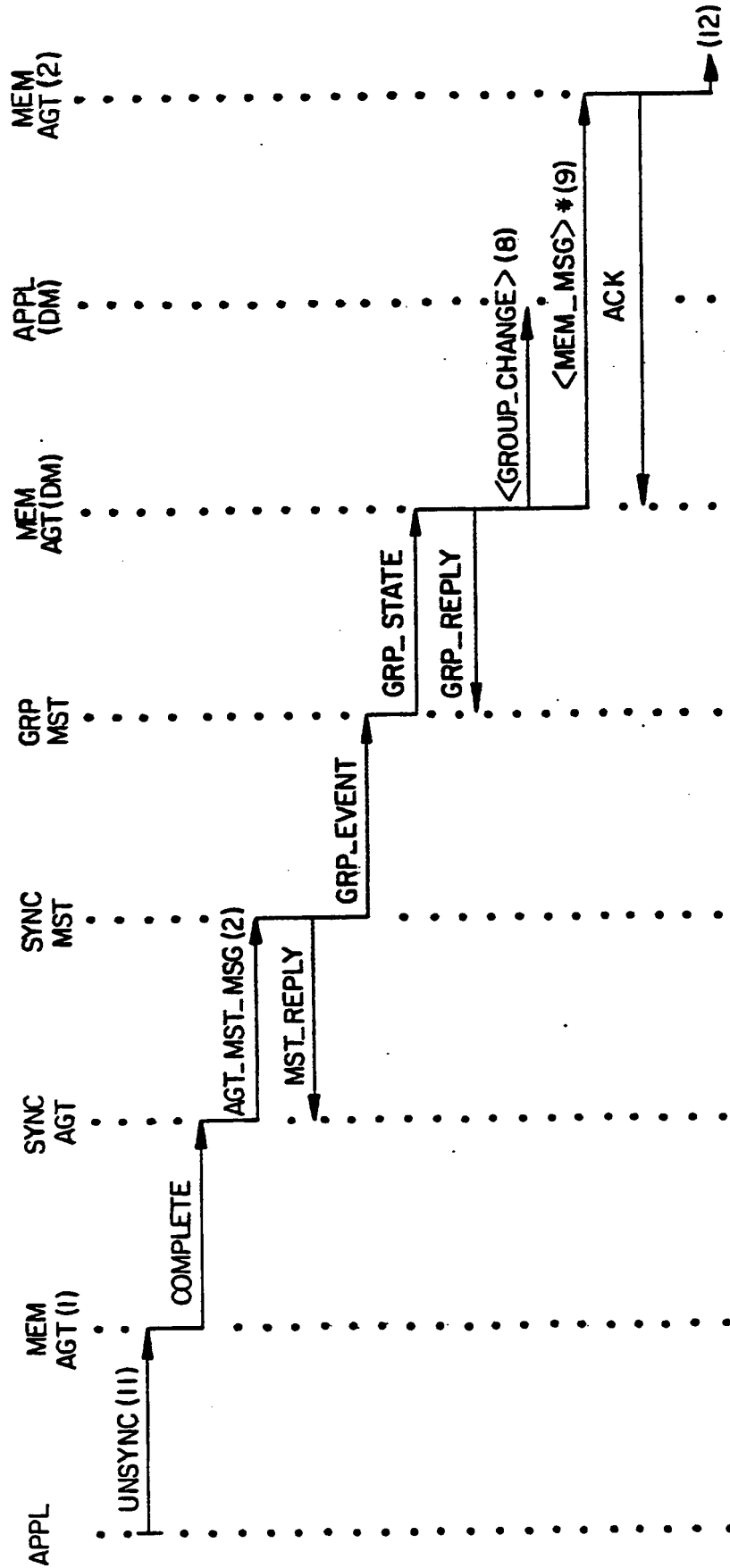


FIG. 8

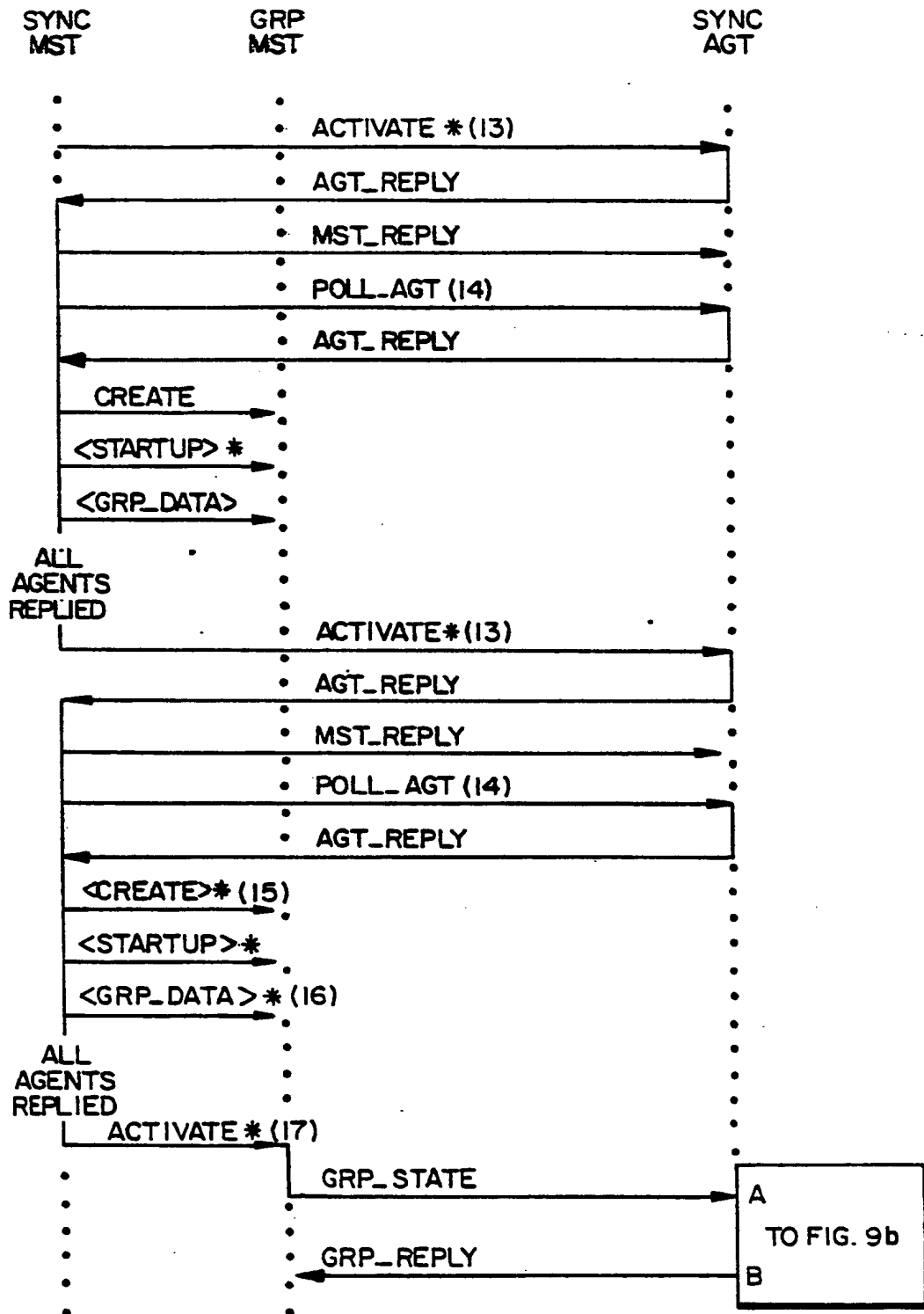


FIG. 9a

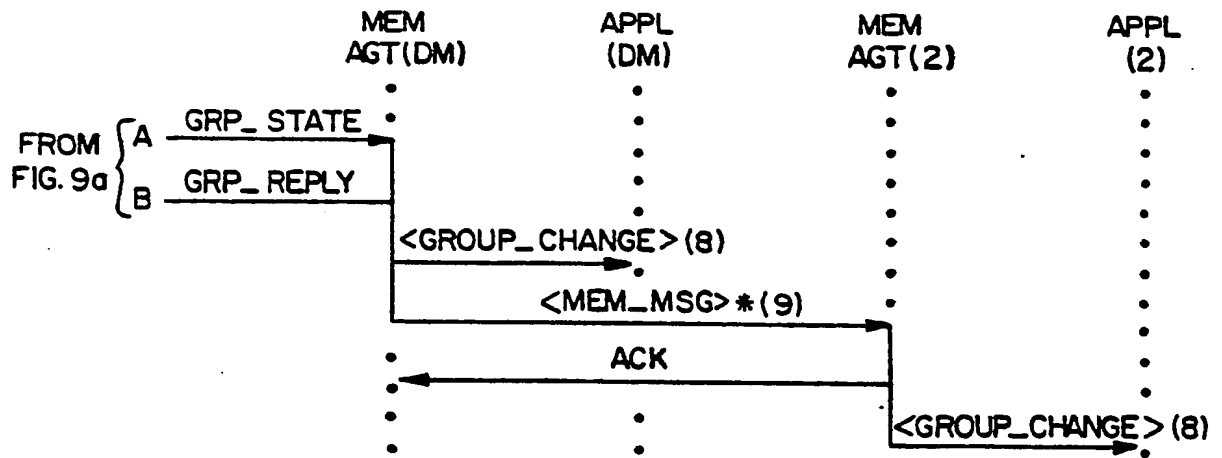


FIG. 9b

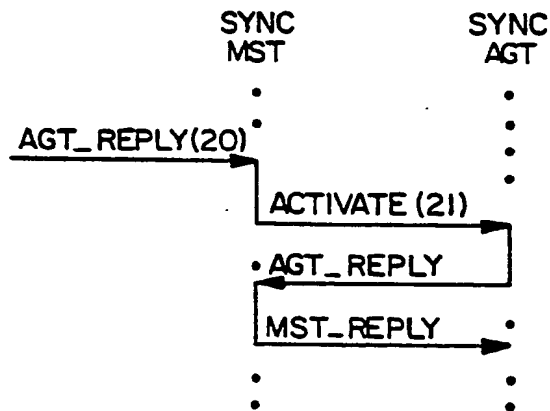


FIG. 11

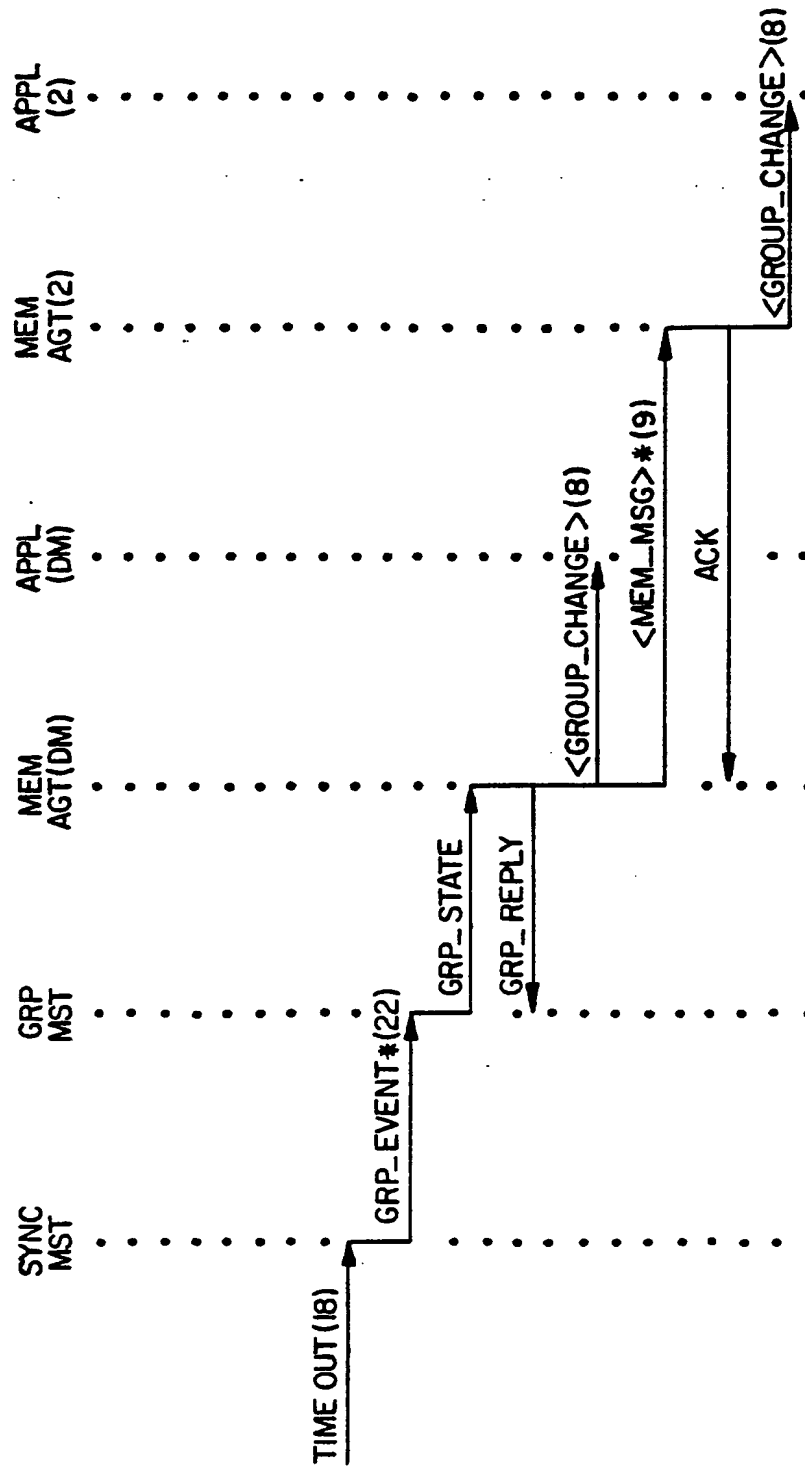


FIG. 10

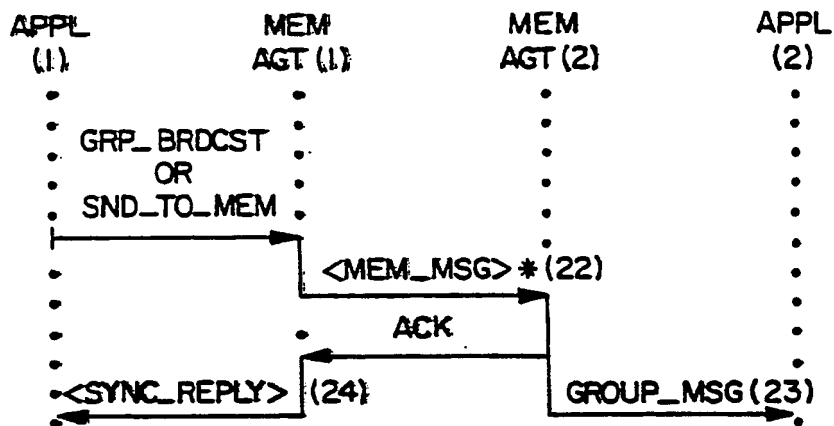


FIG. 12

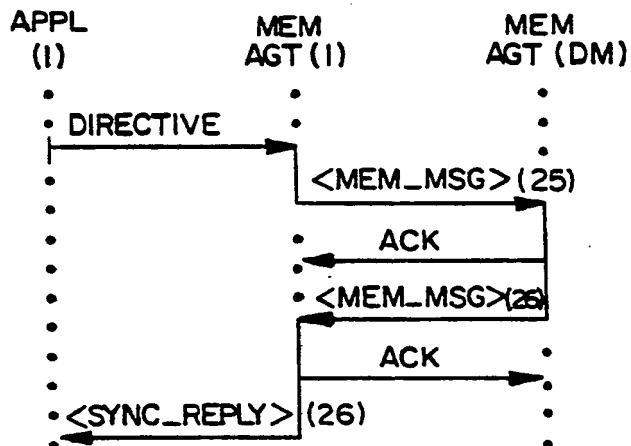


FIG. 13

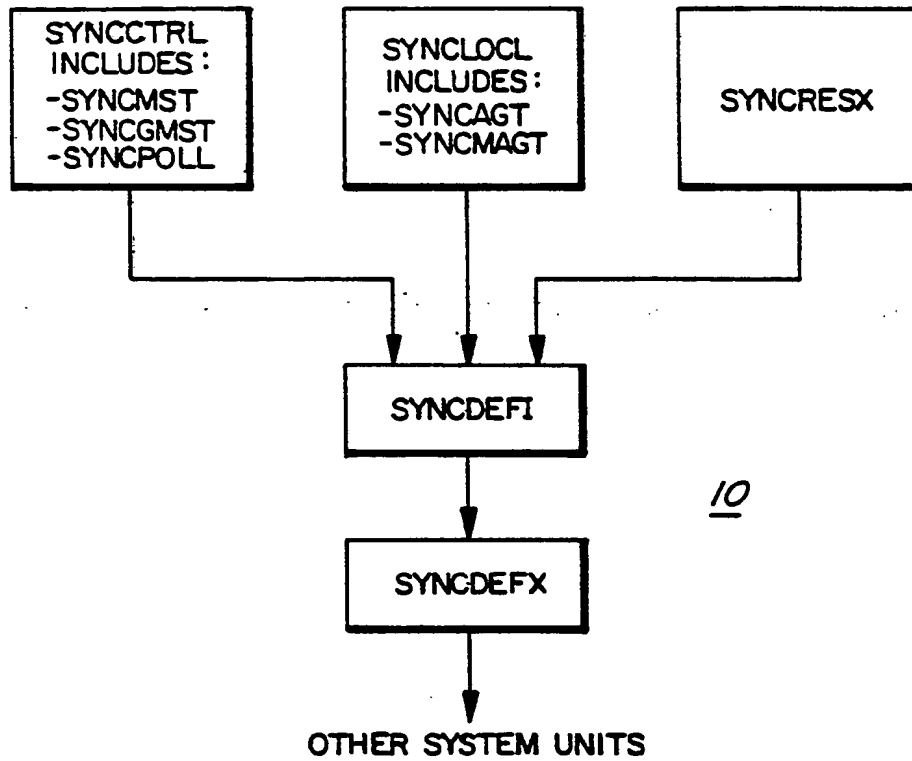


FIG. 14



(19)



Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number:

**0 251 584
A3**

(12)

EUROPEAN PATENT APPLICATION

(21) Application number: 87305416.7

(51) Int. Cl.⁵ G06F 9/46

(22) Date of filing: 18.06.87

(30) Priority: 26.06.86 CA 512479

(43) Date of publication of application:
07.01.88 Bulletin 88/01(84) Designated Contracting States:
AT DE FR GB IT NL SE(88) Date of deferred publication of the search report:
25.04.90 Bulletin 90/17(71) Applicant: NORTHERN TELECOM LIMITED
600 de la Gauchetiere Street West
Montreal Quebec H3B 4N7(CA)(72) Inventor: Goyer, Pierre
32 Terrasse David
Gatineau Quebec J8V 1G4(CA)
Inventor: Selic, Branislav Vladeta
7 Whelan Drive
Nepean Ontario K2J 2A3(CA)(74) Representative: Crawford, Andrew Birkby et al
A.A. THORNTON & CO. Northumberland
House 303-306 High Holborn
London WC1V 7LE(GB)

(54) A synchronization service for a distributed operating system or the like.

(57) A synchronization service which can be incorporated into a distributed operating system as a shared service. It allows the realization of different custom-built synchronization strategies for different applications. This approach is based on defining a general set of application-independent synchronization primitives. These are provided by the distributed operating system in the form of a synchronization service. By themselves the individual primitives are insufficient to provide synchronization. However, they can be combined in different ways to realize customized synchronization strategies. This leaves the ultimate responsibility for synchronization with the application, but in a much simplified form. Application programs can combine these primitives to construct the most suitable form of synchronization.

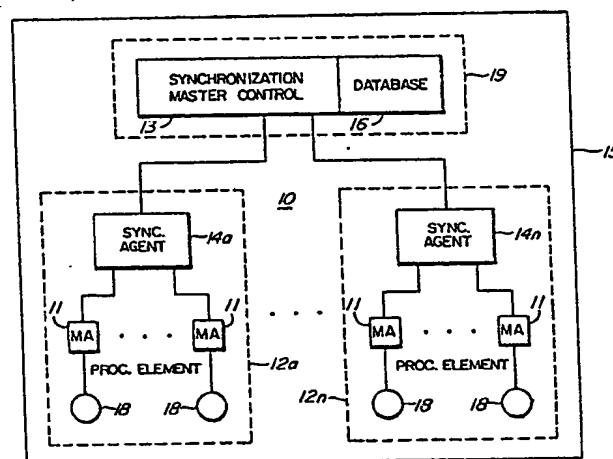


FIG. 1

EP 0 251 584 A3



European Patent
Office

EUROPEAN SEARCH REPORT

Application Number

EP 87 30 5416

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.4)
A	EP-A-0 035 778 (BOEING) * Page 13; page 14, lines 1-7; page 15; pages 25-29; figures 1,3 * ---	1,5,8, 14,19	G 06 F 9/46
A	THE FIFTH GENERATION CHALLENGE - PROCEEDINGS OF THE ASSOCIATION FOR COMPUTING MACHINERY 1984 ANNUAL CONFERENCE (ACM'84), San Francisco, CA, 8th-14th October 1984, pages 179-188; J.C. BROWNE et al.: "Zeus: An object-oriented distributed operating system for reliable applications" * Page 182, column 2, lines 33-59; page 184, paragraph 3.2.1 * ---	5,8,14, 19	
A	PROCEEDINGS OF THE TENTH ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, Washington, 1st-4th December 1985, pages 79-86; K.P. BIRMAN: "Replication and fault-tolerance in the ISIS system" * Page 82, column 1, lines 1-17,35-43; page 83, column 2, lines 51-57; page 84, figure 1 * ---	3,7,10	
A	US-A-4 145 739 (WANG LAB) * Column 2, lines 32-44; figure 3; abstract * -----	19	
The present search report has been drawn up for all claims			
Place of search THE HAGUE		Date of completion of the search 18-01-1990	Examiner DHEERE R.F.B.M.
CATEGORY OF CITED DOCUMENTS X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document		T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons ----- & : member of the same patent family, corresponding document	

EPO FORM 150 (03.82 (P0401))

XP-002151749

Technical Disclosure Bulletin

Vol. 30 No. 10 March 1988

p146-148

P.D. 1-03-1988	3
P. 146-148	

G 06 F 9 / 46 R 6

LOW-LEVEL INTERPROCESS COMMUNICATION FACILITY INTERFACE

Word 1	Data Flags	Ref Bit	Data Length
Word 2	Data Offset/Referenced RID		
Word 3	Segment Identifier		

An implementation of several architected inter-process communication facility (IPCF) verbs (interface commands) can be designed as a two-tiered interface. A high-level interface is used by untrusted users and users in an unpinned environment. The high-level interface to each verb provides the architected interface, validates the input to IPCF, and performs any pinning of data in storage that is needed before the use of the underlying transport mechanism. A low-level interface is used both by trusted IPCF users running in a pinned environment and by the high-level portion of the implementation of that verb. The low-level interface to each verb accepts as input a modified version of the architected interface and provides its users with a fast path to the transport mechanism.

The Inter-Process Communication Facility (IPCF) is a set of functions, called verbs, which provides two processes the ability to communicate with one another using architected interfaces. IPCF may use a number of different underlying transport mechanisms to support this communication, depending on what transport mechanisms are available, and whether the communicating processes are running on the same processor or on different processors connected by one of the transport mechanisms. An IPCF connection between two processes is opened using the IPCF Open verb, then requests and responses to those requests are sent on this connection using other IPCF verbs. Associated with the verbs which concern requests are Data Descriptor Elements (DEs). DEs are fields in the Operation Request Block (ORB), the architected interface to IPCF, which describe the data associated with the request. For example, if the process which sends a request (the requester) needs to send some data to the process which is to perform the request (the server), DEs are placed in the ORB to describe this data to the IPCF and ultimately to the underlying transport mechanism, so that the data can be used by the server.

LOW-LEVEL INTERPROCESS COMMUNICATION FACILITY INTERFACE - Continued

The IPCF verbs which have the longest potential path lengths, and which are most often used, are those verbs for which the user may specify DEs: Send Request, Receive Queue, Receive Request, Receive Data, and Send Response. For these verbs, the greatest contribution to path length is in the processing of the DEs. Thus, while the interface control block to the low level remains the Operation Request Block (ORB), the DEs associated with the ORB must be changed to a new, easier-to-process format, as depicted in the figure.

Data Flags, Word 1, Bytes 0,1 for Request Response Control Block (RRCB) Descriptor Elements. The IPCF implementation could either be responsible for setting the value of the Bus Unit field (Byte 1 bits 3-7), or it could require that the low-level user know this value and set it.

Reference Bit, Word 1, Byte 2, bit 0 : if this bit is set, then this DE is a Reference DE and Word 2 of this DE should be interpreted as the Request ID (RID) of the referenced request.

Data Length, Word 1, Byte 2, bit 1-7, Byte 3 : This field specifies the length, in bytes, of the data in the field specified by the following offset field. This field is reserved and must be zero if the Reference Bit is set to one (1). This format restricts the amount of data described by one DE to 32K-1 bytes, but a further restriction that the data must fit within a page makes this no problem. If pages ever were larger than 32K-1 bytes, the Reference bit would have to be moved.

Data Offset/Referenced RID, Word 2 : If the Reference bit is set to one (1), this word contains the RID of the referenced request. Otherwise, this is the offset within the memory segment identified in Word 3 of the data being described.

Segment Identifier, Word 3 : If the Reference bit is set to one (1), this field is reserved and must be zero. Otherwise, this is the Segment Identifier of the data being described.

The importance of the Offset and Segment ID is that they uniquely identify a real address which IPCF can determine and place in an RRCB if necessary. In some implementations, the meanings of words 2 and 3 when the reference bit is zero might change, but they must uniquely define a real address.

These further restrictions also apply:

1. No data described in, or referenced by, a DE may cross a page boundary. Thus, the value in the Data Length field of a DE will always be less than or equal to the page size.

LOW-LEVEL INTERPROCESS COMMUNICATION FACILITY INTERFACE - Continued

2. All data described in, or referenced by, a DE must be pinned.
3. All DEs are assumed to be error free and in the architected sequence.
4. No combined verb functions are implemented at the low level.

This Page Blank (uspro,

EUROPEAN PATENT APPLICATION

Application number: 89116585.4

Int. Cl.⁵ **G06F 13/38** , **G06F 13/28**

Date of filing: 08.09.89

Priority: 14.09.88 US 244920

Date of publication of application:
21.03.90 Bulletin 90/12

Designated Contracting States:
DE FR GB IT NL

Applicant: **NATIONAL SEMICONDUCTOR CORPORATION**
2900 Semiconductor Drive P.O. Box 58090
Santa Clara California 95051-8090(US)

Inventor: **Michael, Martin S.**
2301 Flint Avenue
San Jose, CA. 95148(US)

Representative: **Sparing - Röhl - Henseler**
Patentanwälte
Rethelstrasse 123 Postfach 14 02 68
D-4000 Düsseldorf 1(DE)

Universal asynchronous receiver/transmitter.

Data characters to be transferred from a peripheral device to a central processing unit are serially shifted into the receiver shift register of a universal asynchronous receiver/transmitter (UART). A multiple byte first-in-first-out memory stores a plurality of data characters received by the shift register. The UART checks the status of each data character stored in the FIFO to determine whether it will trigger an exception. A bytes till exception register indicates the number of data characters remaining in the FIFO until an exception is encountered. Then, upon request by the CPU, the UART provides the count of consecutive valid data characters from the top of the FIFO to the first exception, eliminating the need to check status on every transferred byte. Each of the multiple channels of the UART includes an Initialization Register. Setting the appropriate bit Initialization Register of any UART channel allows concurrent writes to the same selected register in each channel's register set. This function reduces initialization time for all of the common parameters that are loaded into each channel's registers. The UART implements a methodology which allows for the processing of any control characters or errors received by the UART during DMA while internal and/or external FIFOs are being used.

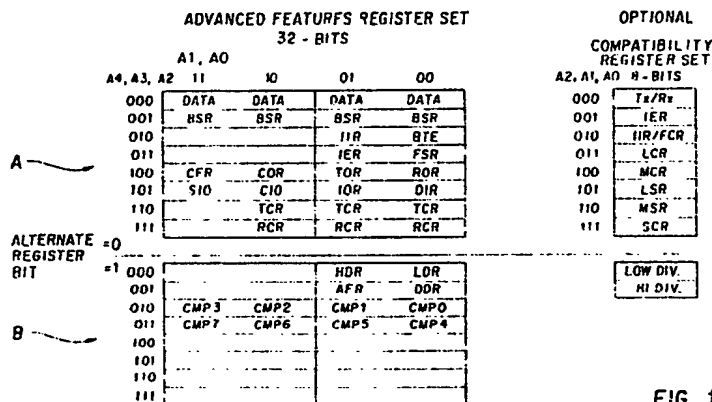


FIG. 1

REGISTER SET OVERVIEW (8, 16, 32 BITS)

Xerox Copy Centre

EP 0 359 137 A2

IMPROVED UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER

The present invention relates to data communications between a data processing system and its associated peripheral devices and, in particular, to an improved universal asynchronous receiver/transmitter (UART) device. According to a first aspect of the invention, multiple data bytes can be transferred from the UART in a single access without checking the error status of each individual byte transferred. According to
 5 a second aspect of the invention, the multiple-channel UART allows concurrent writes to the same register of each channel's identical register set. According to a third aspect of the invention, the UART processes any control characters or errors received by the UART during a direct memory access (DMA) transfer while either internal or external first-in-first-out (FIFO) memories are being used.

Data communications is a broad term used to define the transmission of data from one point to another.

10 To ensure coherent data communications between two or more points, e.g. between a data processing system and one of its peripheral devices, an interface standard is established to define the characteristics of the communication link.

The most popular interface standard for data transmission is asynchronous communication. This standard specifies that each data character to be transmitted be preceded by a "start" bit and be followed
 15 by one or more "stop" bits. Between characters, a mark condition is continuously maintained. Because each transmitted character is bracketed by these "start" and "stop" bits, the receiver is resynchronized with each transmission, allowing unequal intervals between characters.

One commonly used asynchronous data communications device is the Universal Asynchronous Receiver Transmitter, or UART. A conventional UART relies on two separate serial shift registers, each with
 20 its own serial port and clock, to receive data from and transmit data to a modem or peripheral device in response to control signals from the associated data processing system. This architecture allows data to be simultaneously sent and received through the UART at different data rates.

To transmit data from its associated data processing system to a selected modem or peripheral device, a UART can request the parallel transfer of data (typically an 8-bit character, or byte, which is placed on the
 25 system's data bus) into the UART'S transmitter holding register. The transmitter holding register then transfers the data to a transmitter shift register which serially transmits each bit of data to the peripheral device. Initially, when the transmitter holding register is empty, the UART signals the CPU that it is ready to receive data. Data is transferred when a data strobe input from the systems to the UART is appropriately pulsed.

30 Since the transmitter holding register is "empty" as soon as the parallel transfer of data to the transmitter shift register occurs, even if the actual serial shifting of data by the shift register is not complete, the UART can indicate to the data processing system that a new data character may be loaded into the holding register. When the new data is loaded into the holding register, if the serial transmitter shift register is not yet free, then the data is held in the holding register until the serial shift of the initial data is
 35 completed. The transfer of the new data into the shift register is then allowed to take place.

Thus, a conventional UART can retain a maximum of two data characters for transmission from its associated data processing system. If the full transmission requires the transfer of more than two characters, then the data processing system, which can transfer data much faster than the UART'S transmitter shift register, must either wait for the shift register to complete its serial transfer or undertake
 40 different tasks and then respond to multiple interrupts from the UART to complete the transmission. Both alternatives are an extremely inefficient use of data processing time.

Receipt of data by the data processing system from a modem or other peripheral device via the UART is subject to the same time inefficiencies as is data transmission. That is, the processor is inhibited by the operating rate and data capacity of the UART'S receiver section. As in data transmission, to receive data,
 45 the UART utilizes a shift register and a holding register. A data character is shifted serially from the modem or peripheral device into a serial-to-parallel receiver shift register. When the entire data character has been assembled in the shift register, it is transferred to a receiver holding register, freeing the receiver shift register to receive the next character from the transmission line. The UART indicates to the processor system that it has received data ready to be transferred and places the data on the system bus for parallel
 50 transfer when the appropriate strobe is received from the system.

UARTs may be used either in an interrupt mode or in a polling configuration. In the interrupt configuration, the UART sends an interrupt to the data processing system which services it by either placing data on or retrieving data from the system bus. Because a conventional UART can only retain a single data character in each of its receiver and transmitter holding registers, multiple interrupts are required if many data bytes are to be transmitted or received.

To reduce the interrupt overhead of the processor, a more recent UART design has replaced the single-byte receiver and transmitter holding registers with multiple-byte first-in-first-out (FIFO) memories. The National Semiconductor Corporation NS16550A UART utilizes two user-selectable 16-byte FIFO memories as transmitter and receiver buffers. These transmitter and receiver FIFOs permit accumulation of data characters within the UART, eliminating the requirement for multiple interrupts to the processor in its transmission and receipt of data. A UART of this type is described in pending U.S. Patent Application Serial No. 924,797 filed October 30, 1986 by Michael et al for ASYNCHRONOUS COMMUNICATION ELEMENT; the just-identified Michael et al application is hereby incorporated by reference to provide additional background information for the present invention.

Although the NS16550A UART is a highly advanced device, its status indications are based primarily on single byte error indications.

In the vast majority of cases, data that has been received by the UART is error free. Conventional status indications, however, have not allowed the data processing system to detect the number of consecutive error-free data bytes in the receiver FIFO. This prevents the removal of consecutive data bytes by the data processing system until the status for each byte is first read. Since the status of error-free data is inconsequential, a considerable portion of the access time of the data processing system is being wasted, i.e. two clock access for each data byte to be read in the case of error-free data.

UARTs are also available that provide multiple channels for asynchronous communications between a data processing system and a number of associated peripheral devices. Each UART channel includes its own register set, identical to the register set of each of the other channels. This register set stores information that characterizes the channel relative to its operation. Typically, the register set of each channel is loaded with the appropriate digital information on initialization of the UART; this information may then be modified dynamically to meet changing operating requirements. A problem inherent in conventional multi-channel UARTs is that, although it may be desired to load identical information into corresponding registers of each channel, this information must be sequentially loaded into the registers of individual channels. Thus, the loading of the registers with identical information, both upon initialization and upon dynamic modification wastes valuable processing time.

The transfer of data in a data processing system can be generally referred to as one of three basic types: I/O mapped, memory-mapped or direct memory access (DMA). I/O-mapped and memory-mapped transfers require processor intervention, thus tying up the processor during the time that data transfers are being implemented. In DMA transfers between a peripheral device and system memory, a path is provided for direct data transfer without processor intervention. Thus, utilizing this path, the peripheral device can transfer data directly to or from the memory at high speed while freeing the processor to perform other tasks during the transfer.

Unfortunately, the DMA capabilities of conventional UARTs do not allow for the handling of control characters, errors or varying amounts of data received by the UART during DMA while using either internal or external FIFOs. Conventional UARTs do not distinguish between control characters, errors or varying amounts of valid data before requesting a DMA transfer. Thus, the data processing system must either resolve all exceptional data cases or prevent DMA transfer of received data. However, as stated above, since most of the received data are valid and without exceptions, the UART need only request processing time for data transfers when it detects an exceptional data byte.

A Universal Asynchronous Receiver/Transmitter (UART) in accordance with the present invention provides an associated data processing system with a count of the number of consecutive data bytes that can be removed from the UART'S receiver first-in-first-out (FIFO) memory before an exception must be handled. The count is provided to the data processing system on each read cycle. The processor then proceeds to read all consecutive valid data bytes in a single access. The number of clock cycles required for the read access equals one cycle for reading the valid bytes count plus one cycle for each byte to be read. This represents a considerable savings in processor time, since conventional UARTs require that the processor access two registers, status and data, for every data byte removed from the receiver FIFO.

Thus, in accordance with this first aspect of the invention, data characters to be transferred from a peripheral device to a central processing unit are serially shifted into the receiver shift register of a universal asynchronous receiver/transmitter. A multiple byte first-in-first-out memory stores a plurality of data characters received by the shift register. The UART checks the status of each data character stored in the FIFO to determine whether it will trigger an exception. A "bytes till exception" register indicates the number of consecutive data characters remaining at the top of the FIFO until an exception is encountered. Then, upon request by the processor, the UART provides the count of consecutive valid data characters from the top of the FIFO to the first exception, eliminating the need to check status on every transferred byte.

In accordance with a second aspect of the present invention, each of the identical register sets

associated with each of the multiple channels of the UART includes an Initialization Register. Setting the appropriate bit in the Initialization Register of any UART channel allows the data processing system to concurrently write to the same selected register in each channel's register set. This function reduces initialization time for all of the common parameters that are loaded into each channel's registers.

In accordance with a third aspect of the present invention, the UART implements a methodology which allows for the processing of any control characters, errors or varying amounts of other data received by the UART during DMA while internal and/or external FIFOs are being used.

Other features and advantages of the present invention will be understood and appreciated by reference to the detailed description of the invention provided below which should be considered in conjunction with the accompanying drawings.

Figure 1 is a schematic representation of an Advanced Features Register set of a UART in accordance with the present invention.

Figure 2 is a block diagram illustrating the architecture of a UART in accordance with the present invention.

Figure 3 is a schematic representation of Rx FIFO and Tx FIFO data transfers in a UART in accordance with the present invention.

Figure 4 is a schematic representation of the format of Data, Byte Status, Channel Status and Channel Exception Registers of the Advanced Features Register Set of a UART in accordance with the present invention.

Figure 5 is a schematic representation of the format of Control, Channel I/O, Tx CNT and Rx CNT Registers of the Advanced Features Register Set of a UART in accordance with the present invention.

Figure 6 is a schematic representation of the format of Divisor, Initialization, Comparison Registers 0-3 and comparison Registers 4-7 of the Alternate Register Set of a UART in accordance with the present invention.

Figure 7 is a flow sheet illustrating the procedure for maintaining an internal and/or external transmitter FIFO in conjunction with a system DMA unit.

Figure 8 is a flow sheet illustrating the procedure for maintaining an internal and/or external receiver FIFO in conjunction with a system DMA unit.

Referring to Fig. 1, the advanced register set A of a Universal Asynchronous Receiver/Transmitter (UART) in accordance with the present invention consists of eight 32-bit wide registers including five registers (addresses 00000 through 10000) for standard serial channel operation, DMA operation, and automated transmit Tx flow control, one register (address 10100) for modem or general purpose I/O features and two registers (addresses 11000 and 11100) for external FIFO control.

In addition to the advanced register set A, an alternate register set B that includes eight 32-bit registers is accessible by setting an Alternate Register Bit. The alternate register set B is used primarily during UART initialization: Received data comparison registers, a baud rate divisor, and an I/O data direction register (accessed at addresses as indicated in Fig. 1) are all accessible when the Alternate Register Bit is set.

Referring to Fig. 2, five address pins A0-A4 are used to select the internal registers. Identical register sets are present on each of four serial channels available in the UART embodiment described below; channel select pins are provided for access to the register sets of each serial channel.

All four UART channels reset to a Compatibility Mode. Two bits in a Compatibility Mode register set, shown in Fig. 1, allow selection of 8, 16, 32-bit bus width or Compatibility Mode operation. These two bits are IER6 and IER7 in the Interrupt Enable Register (IER), which is discussed in greater detail below. If IER6,7 are set to 00, 01, 10 to 11, then the UART modes are Compatibility, 8, 16, or 32-bit, respectively.

Referring again to Fig. 1, a DATA Register (address 00000) is organized as 4, 2 or 1 byte wide fields, depending on the programmed bus width. In Fig. 2, the DATA Register represents the storage location at the top of a FIFO, in the case of both the receiver and transmitter portions of the UART, and is used to read data from a Receiver Rx FIFO 12 or write data to a Transmitter Tx FIFO 14. (Any reference in the following description to "RX FIFO" or "Tx FIFO" is a reference to internal UART FIFOs 12 and 14, respectively. Any reference to an external Tx FIFO or an external Rx FIFO will be preceded by the word "external.")

The Rx and Tx FIFOs 12 and 14, respectively, are a constant length regardless of the bus width. This means that one-half the number of accesses are required to transfer data to/from these FIFOs in the 16-bit mode as compared to 8-bit mode and one-quarter the number of accesses are required to transfer data to/from these FIFOs in 32-bit mode as compared to 8-bit mode.

As illustrated in Fig. 3, when transferring multiple bytes from Rx FIFO 12 through a Data Bus Buffer 10 during a single access (16-bit or 32-bit Mode), the least significant byte on the CPU bus is the byte that was received earlier than the other bytes; when transferring multiple bytes to Tx FIFO 14 through Data Bus

Buffer 10 during a single access (16-bit or 32-bit Mode), the least significant byte on the CPU bus is the first byte sent out on the serial line.

Byte Status Register (BSR) 16 provides the associated line status and byte match information for each byte that the CPU reads from Rx FIFO 12. As shown in Fig. 1, BSR 16 (address 00100) contains 4, 2 or 1 byte-wide fields, depending on the programmed bus width. The type of information provided by BSR 16 for each received data byte is overrun, parity and framing errors; break indication, byte match, and byte match ID. Individual byte status remains in BSR 16 until the associated data has been read from Rx FIFO 12. BSR 16 is updated with status for the next group of data bytes as soon as these bytes can be read from the Rx FIFO 12. In the block mode, described in greater detail below, BSR 16 accumulates the status of each byte until it is read. If there are fewer data bytes than the full bus width to be given during the data read, the status bytes in BSR 16 will correspond to the position of the data bytes that can be read from Rx FIFO 12.

A Channel Status Register (CSR) provides the status indication for all interrupts conditions. As shown in Fig. 1, CSR (address 01000) contains two byte-wide fields named Interrupt Identification (IIR) and Bytes Till Exception (BTE) and two byte-wide reserved fields.

The bits of the IIR field are set when their associated interrupt condition is active. The appropriate bit in the IER must be set before any indication in the IIR field can activate the external interrupt signal. The IIR bit, however, is set when there is an active interrupt condition regardless of the interrupt enable bit setting.

The following interrupt conditions each set one IIR bit:

1. Reaching a programmed receiver trigger level or an active receiver timeout condition sets bit IIR7.
2. A match in any of the comparison Registers (described below) sets bit IIR6.
3. A line status error condition (parity, framing, overrun, break) sets bit IIR5.
4. A change in any input status indicator from the general purpose I/O lines programmed as inputs in the Data Direction Register (described below) sets bit IIR4.
5. A match in either of the Comparison Tx Flow Control Registers sets bit IIR3.
6. A Tx FIFO empty (TFE) condition or completing a pre-programmed number of transmitter transfers via DMA sets bit IIR2. The TFE condition is cleared after the transmitter enable bit is reset or a byte is loaded into the Tx FIFO 14.
7. Completing a preprogrammed number of receiver transfers via DMA sets bit IIR1.

IIR bit 0 (TEMT) is set when the transmitter is completely empty. The setting of this bit cannot cause an interrupt to occur, but it is included in this register for the convenience of checking the transmitter during half duplex operation.

Referring back to Fig. 2, in accordance with one aspect of the present invention, Bytes Till Exception Register (BTE) 18 indicates how many bytes remain in Rx FIFO 12 until an exception is encountered. An exception is defined in this context as anything other than valid data, e.g. an empty Rx FIFO 12, a line status error or a Comparison Register match. The BTE count is the status indicator that the system will use most often, since data is usually received by the UART without error. The count in BTE 18 is updated after every read of the Rx FIFO 12 by the CPU.

BTE 18 eliminates the need to check status on every byte, since the CPU can now allow the UART to perform this function. The UART then provides the count of consecutive valid data bytes from the top of the Rx FIFO 12 to the first exception. The CPU uses this variable count to determine the number of bytes to extract from the Rx FIFO 12 in a read access.

In the 16-bit mode, the Channel Status is accessed through one 16-bit wide register; in the 8-bit mode, the Channel Status is accessed through the two byte-wide registers IIR and BTE.

Referring to Fig.1, Channel Exception Register (CER) (address 01100) contains two byte-wide fields named Interrupt Enable (IER) and FIFO Status (FSR) and two byte wide reserved fields.

The IER field holds the interrupt enable data. The eight possible interrupts are:

1. Receiver FIFO Trigger Level (RFT) or Receiver Count (RCR);
2. Match (MCH);
3. Line Status (LSI);
4. Input Status (ISI);
5. Transmitter FIFO Flow Control (TFC);
6. Transmitter FIFO Empty (TFE) or Transmitter Count (TCR); and
7. Receiver Timeout (RTO)

The RFT and RCR interrupts are mutually exclusive operations in the UART, as are the TFE and TCR interrupts. All interrupt priorities are user determined.

The FSR field indicates the number of empty spaces in Tx FIFO 14, whether or not there is a match or an error detected anywhere in Rx FIFO 12, and which bytes in the DATA Register are valid receiver serial data. Bits FSR7,6 indicate the number of empty spaces in Tx FIFO 14 according to the following code:

FSR	7	6	Tx FIFO State
	0	0	empty
	0	1	1/2 full
	1	0	1 space empty
	1	1	full

Bit FSR5 indicates that at least one byte is loaded in Rx FIFO 12 that matches a byte in Comparison Registers 2-7. This bit is used as an advanced signal that there is a control character to be processed.

Bit FSR4 indicates that there is at least one Line Status indication associated with a data byte in Rx FIFO 12. This bit is used as an advanced signal that an error or break has occurred.

Bits FSR3-0 indicate the byte positions at the top of Rx FIFO 12 that contain valid data. This is needed by the CPU only when there is less than an integral number of data bytes presented during a 16- or 32-bit wide read. When there are less bytes of valid receiver data in DATA Register than the data bus is wide, then bits FSR3-0 in the FSR field are set to indicate the valid data byte positions. Only the consecutive data bytes that are valid at the time that the FSR field is read will be issued to the CPU during the next read of DATA Register. Thus, by comparing the bits that are set in the FSR to the bytes received from DATA Register, the user can determine which bytes are valid. This technique means that the UART will not move additional data into the top of the FIFO until the CPU reads the DATA register. Therefore, the CPU should read the DATA Register as soon as possible after reading the FSR to "free" any unused space in the top of the FIFO. Bytes will still be taken into the open space not included in the top of the FIFO during this time.

Using a 32-bit wide CPU data bus, the following example is given. The CPU reads the FSR field when there are only three consecutive valid data bytes in Rx FIFO 12 and no other data. Thus, the FSR has the three lowest order bits set. When the CPU then reads the DATA Register, the UART will only issue three valid data bytes in the lowest byte positions and a 00 character in the highest byte position. This is the case even if a valid data byte enters Rx FIFO 12 between the time the CPU reads the FSR field and the time it reads the DATA Register. At all other times, the data presented to the CPU bus will be as wide as the bus.

Using a 16-bit wide CPU data bus, the following example is given. The CPU reads the FSR field when there is only one valid data byte in Rx FIFO 12 and no other data. Thus, the FSR field has the lowest order bit set. When the CPU then reads the DATA Register, the UART will only issue one valid data byte in the lowest byte positions and a 00 character in the highest byte position. This is the case even if a valid data byte enters Rx FIFO 12 between the time the CPU read the FSR field and the time it read the DATA Register. At all other times, the data presented to the CPU bus will be as wide as the bus.

The Control Register (address 10000) is the heart of UART operations. It contains four byte-wide fields that are identified in Figs. 1 and 2 as Channel Format (CFR), Channel Operations (COR), Transmitter Operation (TOR) and Receiver Operation (ROR).

The CFR field controls the UART block mode enable and serial data format. The block mode determines whether or not receiver errors will be accumulated in BSR 16. If the block mode is enabled, BSR 16 accumulates all errors, breaks and matched information associated with data passing through the top of Rx FIFO 12. The results of this accumulation is indicated by the LSI and Match bits in the IIR field. After the CPU reads BSR 16, all status bits are cleared, including the associated IIR bits. Setting bit CFR6 enables the block mode.

The serial data format specification includes data length (5-8 bits), stop bits (1, 1-1/2, 2), and parity. When bit CFR5 is 0, one stop bit is sent with all data combinations. When bit CFR5 is 1, two stop bits are sent with all combinations of data, except 5-bit data which is sent with 1.5 stop bits. The setting of bit CFR5 does not affect the receiver; it only checks for 1 stop bit. Bits CFR4,3 select the number of data bits transmitted or received in each character. If bits CFR4,3 are 00, 01, 10, or 11, then 5, 6, 7, or 8 bits are serially transmitted and described, respectively. Setting bit CFR2 enables parity transmission and reception. When parity is enabled, even parity is selected by setting bit CFR1. Clearing bit CFR1 when parity is enabled results in odd parity. Setting bit CFR0 enables mark or space parity if the parity enable bit CFR2 is set. If bits CFR2 and CFR0 are set, then CFR1 determines whether mark (CFR1 = 0) or space (CFR1 = 1) parity is transmitted and received.

Bit CFR7 is reserved. When written to, this bit should be set to 0 and when read it will always indicate 0.

The COR field controls Alternate Register selection, receiver coupling, the data bus access width, DMA mode selection, transmitter DMA enable and receiver DMA enable. The Alternate Registers (Baud Rate Divisor, Alternate Function Register, Data Direction Register and the Comparison Registers) are accessible

when bit COR7 is set. As stated above, these Alternate Registers are used primarily during initialization to establish the base hardware configuration of the serial channel.

Two bits are used to determine the receiver coupling (00 = normal, 01 = local loopback, 10 = remote loopback and 11 = echo mode). Selecting normal mode default) transfers all of the data the receiver receives to Rx FIFO 12. Selecting the local loopback mode causes the data transferred to the transmitter to be internally sent to the receiver. Selecting the remote loopback mode causes the transmitter to send every byte the receiver receives, without these bytes entering Rx FIFO 12. Selecting the Echo Mode causes the transmitter to send a duplicate of every valid data byte that is in the top of Rx FIFO 12. No bytes that cause exceptions, i.e. an LSI condition or a Match condition) are echoed. When the next byte to be echoed is a byte with an exception, Echo Mode is automatically disabled by setting COR 6,5 to 00 (normal mode). At that point in time, the CPU should respond to the exception and echo the appropriate(s). The CPU will re-enable to ECHO Mode after the exception is processed.

Using the comparison Registers in conjunction with the Echo Mode and Rx DMA only requires CPU intervention during the receipt of control characters or corrupted data. Because there are many control characters in the ASCII set below 1F hex that do not get echoed, Comparison Register 2 has an additional capability. It can match with any byte entering the Rx FIFO that is less than or equal to its programmed value when the "less-than-or-equal-to" option is specified by setting TOR5. When TOR5=0, Comparison Register 2 will operate on an "equal-to" basis only. Thus, in the case of automatically echoing ASCII characters, the CPU can:

1. Set TOR5;
2. Load 1F hex into CMP2;
3. Load 7F hex into CMP3; and
4. Set COR 6,5 to 11, respectively.

All characters received between the value 20 and 7E hex will be automatically echoed by the UART. All other values would create exceptions, thus signalling the CPU and disabling the Echo Mode.

Bits COR4,3 determine the data bus width. The data bus can be either 8, 16 or 32 bits wide. The 8-bit wide bus has two modes of operation: Compatibility Mode or 8-bit Mode. If bits COR4,3 are signal to 00, 01, 10 or 11, then the operating modes are Compatibility Mode, 8-bit mode, 16-bit mode and 32-bit mode, respectively.

Bits COR2,1,0 control Tx and Rx DMA. Bit COR2 determines the DMA mode; the two options are single-transfer (bit COR2=0) and multi-transfer (bit COR2=1). Setting bit COR1 enables transmitter DMA; setting bit COR0 enables receivers DMA. When either of these options are disabled, their output signals (DMA request) are inactive (high).

The TOR field controls the transmitter and Comparison Register operations. Through this byte, the CPU can:

1. Enable or disable the transmitter;
2. Clear Tx FIFO 14;
3. Program a transmitter break;
4. Enable transmitter control through /CTS;
5. Enable CMP2 to match on either a less than or equal to condition;
6. Enable transmitter flow control through the Comparison Registers;
7. Clear the Comparison Registers; and

The transmitter enable bit allows the CPU to control the transmitter directly and also to override any automatic flow control changes that have impacted the transmitter enable state. Clearing Tx FIFO 14 is useful if a retransmission of data is required. The transmitter break bit (TOR2) sets the SOUT signal high for as long as it equals 1. /CTS Tx flow control enables or disables the transmitter as general purpose I/O bit 0 changes from active to inactive, respectively. Comparison Tx flow control enables or disables the transmitter as Comparison Registers 0 and 1 match the incoming data. These last two transmitter control options automatically set and clear the transmitter enable bit. As stated above, bit TOR5 enables the less-than-or-equal-to option for Comparison Register 2. If TOR5=0, then CMP2 will match only when the byte entering Rx FIFO 12 equals its programmed value. If TOR5=1, then CMP2 will match when the byte entering Rx FIFO 12 is less than or equal to its programmed value. Bit TOR6 enables the general purpose Comparison Registers (COMP0-7). Of these registers, the ones that are loaded with data after a reset or a Comparison Register clear are the ones that are actually compared to the incoming data bytes. TOR6 is used to clear the Comparison Registers without resetting the UART; this bit is self-clearing. This feature can be useful when loading the Comparison Registers with a new set of data that contains fewer bytes than the previous set.

The ROR field controls receiver Rx operations. Through this byte, the user can control the insertion of a

flow control character into the transmitter data stream, the receiver clock source, the number of receive character times delayed until a receiver timeout is issued, set-up the receiver trigger level and clear the Rx FIFO 12.

Writing to ROR7 causes the UART to insert the byte programmed in Comparison Register 1 (i.e. XOFF) into Tx Shift Register 20 as soon as the present character in Tx Shift Register 20 has been sent.

The receiver clock (RCLK) for each of the four UART channels can be independently derived from any one of four sources. Setting bits ROR6,5 to 00, 01, 10 and 11 selects the RCLK source to equal the baud rate generator output, 1/2 the baud rate generator output, 1/4 the baud rate generator output or the signal at the RCLK pin, respectively. After reset, the default value of bits ROR6,5 is 00. These options are provided primarily to allow high-speed transmission of data to a peripheral device which will only be providing low-speed keyboard or flow control data back to the CPU. Secondly, it requires fewer pins to support a UART with multiple receivers. The transmitter is unaffected by this selection and is always driven by the baud rate generator output.

A receiver timeout interrupt is used by each serial channel of the UART to indicate that data is present in its Rx FIFO 12 and that no CPU or serial channel activity has occurred during a specified period of time. This feature activates when there is data in Rx FIFO 12 that can't reach the interrupt trigger level. It ensures that the CPU will get an interrupt indicating the presence of receiver data. The amount of delay before a timeout interrupt is issued is programmable using bits 4 and 3 of the ROR field. The delay is based on the receiver clock and is equal to an integral number of receive character times. If bits ROR4,3 are equal to 00, 01, 10 or 11, then the number of receiver character times delayed before a timeout interrupt is issued is 1, 2, 3 or 4 receive character times, respectively. A reminder timeout interrupt is issued if the following conditions are met:

1. Data is in Rx FIFO 12;
2. The CPU has not accessed Rx FIFO 12 during the timeout period; and
3. No new serial number has entered Rx FIFO 12 during the timeout period.

The receiver timeout interrupt timer is reset whenever a CPU access occurs or a byte is added to Rx FIFO 12. It is started when there is a byte in Rx FIFO 12.

Bits ROR2,1 determine the number of bytes that must be in Rx FIFO 12 before a receiver interrupt is issued. Setting bits ROR2,1 to 00, 01, 10, 11 will result in a receiver interrupt being issued when there are 1, 4, 8 or 15 bytes, respectively, in Rx FIFO 12.

Clearing Rx FIFO 12 is useful when there is an error in the FIFO and it is not desired to extract each byte individually before a retransmission occurs.

In 16-bit mode, the Control Register is accessed through two 16-bit wide registers called the Channel and the Tx/Rx Operation. In the 8-bit mode, the Control Register is accessed through the four byte-wide registers CFR, COR, TOR and ROR.

Referring to Figs. 1 and 2, the Channel I/O register (address 10100) controls all eight UART I/O pins that can be used for a modem interface or as general purpose I/O. The modem interface is configured automatically after reset and provides all of the standard inputs (/CTS, /DSR, /DCD, /RI) and outputs (/RTS, /DTR). The Channel I/O register contains four byte wide fields: Set I/O (SIO), Clear I/O (COR), I/O Registers (IOR), and Delta Input (DIR).

The Set I/O field is used to set individual output pins. Corresponding output pins will be set on the trailing edge of the write strobe when their bits in this field have 1s written to them. These eight bits are write only.

The Clear I/O field is used to clear individual output pins. Corresponding output pins will be cleared on the trailing edge of the write strobe when their bits in this field have 1s written to them. These eight bits are write only.

Simultaneously writing to bits in both the Set and Clear I/O fields that correspond to the same output pin will result in no change in the output pin. Simultaneously setting and clearing different output pins through these fields is allowed. Setting or clearing input pins only sets or clears the corresponding output latch.

The IOR field determines the state of the out-going I/O lines. Writing a 1 to any of the IOR bits sets the corresponding output pin high; writing a zero sets it low. All writes to the IOR field are on a byte wide basis. If the corresponding pin is an input, the 1 is written into the output latch, but it doesn't affect the pin. Bits of this field are both read and write. When reading bits that are designated outputs, the value of the output latch is returned. Reading bits that are designated inputs provides the status of the pins.

The DIR field provides input status information for any of the I/O pins programmed as inputs (a change in any of the input lines sets the corresponding bit in this register). The setting of any delta bits in the DIR field can cause an interrupt to be issued to the CPU if ISI interrupts are enabled in the IER field. These

status bits are read only; writing to them does nothing. When the CTS Tx Flow control bit in the DIR field is enabled, I/O bit 0 is automatically made to look like an input. The line becomes the .CTS input line which enables and disables the transmitter.

After reset, the IOR, DIR and DDR fields are automatically initialized to a condition that assumes the UART is connected to a modem or an Electronic Industries Association (EIA) interface.

In this configuration, the bits are all programmed to 1, assigned specific functions (i.e., CTS, .RTS, etc.), and have a predetermined direction established in the DDR 22. Since this is essentially a general purpose I/O port allowing various configurations, the default pin and bit assignments are used when communicating with a modem or an EIA interface to preserve system compatibility. The default assignments for the bits are as follows:

BIT	IOR	DIR	DDR
0	CTS	.CTS	input
1	DSR	DSR	input
2	RI	.RI	input
3	DCD	.DCD	input
4	RTS	.CTS (delta)	output
5	.DTR	.DSR (delta)	output
6	GENR'L I/O	.RI (delta)	input
7	.GENR'L I/O	.DCD (delta)	input

The trailing edge of the .RI (ring indicator) signal sets the status bit in the DIR field.

In the 16-bit mode, the I/O Register is accessed through two 16-bit wide registers called the Set/Clear and the IOR/DIR Registers. In the 8-bit mode, the I/O fields is accessed through the four byte-wide registers SIO, CIO, IOR and DIR.

Referring to Figs. 1 and 2, the TCR and RCR Registers 24 and 26, respectively, contain four byte wide fields, three of which are named Tx Count (TCR) or Rx Count (RCR), respectively. These registers, along with the DMA capability of the UART, can be used to create and control external FIFOs for the data, as described in greater detail below.

The TCR Register 24 (address 11000) is a programmable counter used to track the number of bytes loaded into Tx FIFO 14 via DMA. The number of bytes that are to be transferred to Tx FIFO 14 via DMA before a TCR interrupt is issued by the UART is programmed. The TCR counter 24 decrements for each byte transferred to Tx FIFO 14. When the zero count is reached, transmitter DMAs are disabled and the TCR bit in the IIR field is set. If the TCR bit in the IER field is enabled, then the interrupt pin goes active low. If the TCR bit in the IER field is not enabled, then only the TCR bit in the IIR field is set. The actions associated with the TCR counter 24 only take place if transmitter DMA is enabled and the counter 24 is loaded with a value other than zero. If the TCR counter 24 is not loaded after reset, or is loaded with a value of zero, then the actions associated with this counter do not take place. This DMA process can be halted at any time by disabling transmitter DMA or by loading counter 24 with zero.

The RCR register 26 (address 111) is a programmable counter used to track the number of bytes loaded into system memory (RAM) via DMA. The number of bytes that are to be transferred from Rx FIFO 12 via DMA before receiver DMA interrupt is generated by the UART is programmed. The RCR counter 26 decrements for each byte transferred from Rx FIFO 12. When the zero count is reached, the UART disables receiver DMAs. The Rx DMA bit in the IIR field is set and the UART's INTR line goes active low if it is enabled in the IER field. The actions associated with counter 26 only take place if receiver DMA is enabled and counter 26 is loaded with a value other than zero. If the counter 26 is not loaded after reset, or is loaded with a value of zero, then the actions associated with counter 26 do not take place. This DMA process can be halted at any time by disabling receiver DMA or by loading counter 26 with zero.

Referring again to Figs. 1 and 2, the Baud Rate Divisor Register (address 00000 in the Alternate Register Set) contains two byte wide fields named High Divisor (HDR) and Low Divisor (LDR) and two byte wide reserved fields. The HDR and LDR fields hold the 16-bit wide divisor for the channel's baud rate generator 24. The HDR field contains the high byte of the divisor and the LDR field contains the low byte. This register resets to 000C hex. This is the divisor for 9600 baud if XIN is connected to an 1.8432 MHz clock input.

The Initialization Register (address 00100 in the Alternate Register Set) contains two byte-wide fields named Alternate Function (AFR) and Data Direction (DDR) and two reserved fields. The AFR field contains

three active bits. In accordance with a second aspect of the present invention, setting bit AFR0 in any of the UART's four channels allows the CPU to concurrently write to the same selected register in each channel's register set. This function reduces initialization time for all of the common parameters that are loaded into each channel's registers. The CPU can set or clear bit AFR0 by accessing any channel's register set. When bit AFR0 is set, the channel select pins still determine which channel will be accessed during read operations. Setting or clearing bit AFR0 has no effect on read operations.

Bit AFR1 determines the function of the MF pin. If AFR1 = 0 (the default state), then the MF pin will provide the normal ring indicator function (the signal ring indicator is normally activated by a modem that has sensed an incoming phone call.) When AFR1 = 1, the MF pin will provide the output for that channel's BAUDOUT signal.

Setting bit AFR7 resets the associated channel. This is a user reset that is channel specific, as opposed to the reset that resets all of the channels.

The DDR field establishes the data direction for the general purpose I/O. Setting any bit to 1 in the DDR field causes the corresponding I/O bit to be an output. This field is always accessed on a byte wide basis. After reset, the DDR field is automatically initialized to the default modem configuration. The six lower bits in the DDR field are used in the modem interface and function as previously described, and the two upper bits are programmed as inputs. The DDR field resets to 0c, which establishes the direction for the standard modem and EIA interface signal assignments.

The Comparison Registers (addresses 01000 and 01100 in the Alternate Register Set) are byte-wide registers that are compared to the bytes in Rx FIFO 12 for a match. Both registers consist of four byte-wide fields named Comparison 0 (CP0) through Comparison 3 (CP3) and Comparison 4 (CP4) through Comparison 7 (CP7). These registers are programmed during initialization with the data (typically, hex representation control characters) that are to be matched during receiver operation. Once they are enabled, by setting bit COR5, they will be compared to the received data when it enters Rx FIFO 12. Anytime a match with a received byte occurs, bit 6 in the IIR field will be set when the matched byte is at the top of Rx FIFO 12. An interrupt will be issued to the CPU if it is enabled in the IER field. The Match status bit in BSR 16 associated with the matched data byte will be set, so that the matched byte can be identified. Also, the Comparison Register ID bits in the appropriate BSR 16 will be set. Only Comparison Registers that have been loaded by the CPU since the last reset or since the activation of the Comparison Register clear bit are compared to the received data stream.

Flow control of the transmitted data stream can be controlled by the UART directly. This is done by programming Comparison Register 0 and 1 with flow start and stop characters, respectively. When either of these programmed start or stop flow control characters are matched in Rx FIFO 12, the UART automatically enables or disables the transmitter. The transmitter will complete the sending of any byte in its Tx shift register 20 as the flow control stops characters from being transferred from Tx FIFO 14 to the shift register 20. These Flow Control Comparison Registers enable and disable the transmitter by changing the state of the transmitter enable bit. The flow control characters are received on the SIN line to Rx shift register 26 and can be 8 byte characters. The automatic flow control feature can be overridden at any time by the CPU via the transmitter enable bit. This feature can be disabled via the automatic flow control bit to permit binary file transfer. After reset, Comparison Registers 0 and 1 are automatically loaded with the XON character (11 hex) and the XOFF character (13 hex), respectively. If the Comparison Flow Control bit is enabled, then the matched characters are not put into Rx FIFO 12 and the TFC status bit in the IIR is set immediately. If the Comparison Flow Control bit is not enabled, then the matched characters are put into Rx FIFO 12 and when they reach the top of FIFO 12, they set the TFC status bit in the IIR.

Automatic flow control is also available through the /CTS pin. When enabled, this feature starts and stops transfers to the Tx shift register 20 when /CTS is active or inactive, respectively. Transmitter flow control via /CTS is accomplished by setting or clearing the transmitter enable bit. Transmitter flow control via both the Comparison Registers and /CTS is allowed. If this is the chosen mode of operation, then the most recent input to the transmitter enable bit from the Comparison Registers, the /CTS pin or the CPU prevails.

In 16-bit mode, the Comparison Registers are accessed two at a time. In 8-bit mode, the Comparison Registers are accessed one at a time.

Each of the four serial channels of the UART can control both internal and external FIFOs for the UART transmitter. The internal Tx FIFO 14 for each channel is 16 bytes deep. Based on 24-bit wide registers, the external Tx FIFOs can be up to 16.777216 Mbytes deep.

In an internal Tx FIFO transfer, the CPU writes data to Tx FIFO 14 in bus wide groups. As stated above, the data is sent out serially by the transmitter with the least significant byte sent first. The transmitter sends all data in Tx FIFO 14 as long as Tx FIFO 14 is enabled. If Tx FIFO 14 is disabled while a byte is being

shifted out, that particular byte is finished, but no more bytes are transferred to the Tx shift register 20.

If the number of data bytes to be transferred to Tx FIFO 14 is less than one bus width, the CPU will send advanced action to the UART. As stated above, this is done by activating the /HBE and A0 input signals. Data in the Tx FIFO 14 is always stored in consecutive byte locations regardless of the number of bytes in each CPU transfer.

The CPU must program bits 2 and 1 of the Channel Operation Register (COR2.1) for DMA mode and Tx DMA enable, respectively. Transfers to Tx FIFO 14 executed by DMA are started when Tx FIFO 14 is empty. In DMA mode 0, the transfer request pin deactivates after the first transfer of data into Tx FIFO 14. In DMA mode 1, the transfer request pin deactivates when Tx FIFO 14 is full. All transmitter DMA data transfers must be the full width of the bus or the /HBE and A0 bus signals must indicate a byte-wide transfer.

The UART also provides for maintaining an external Tx FIFO in conjunction with a system DMA unit. This is done to allow increased transmitter FIFO length without significantly increasing UART die size.

Referring to Fig. 7, to maintain an external Tx FIFO, DMA control circuitry requests DMA transfers and notifies the CPU when a preselected number of bytes are to be transferred from the system memory via DMA. One additional register is provided for external FIFO control. This is the Tx DMA counter register. It keeps track of the number of bytes transferred from the memory via DMA. The CPU programs this counter with the number of bytes to be transferred. The counter decrements for each byte transferred during Tx DMA. When the counter reaches 0, the Tx counter bit in the IIR field is set and further Tx DMA requests are automatically disabled by clearing the Tx DMA enable bit (COR1). If enabled, i.e., bit IER2 set, an interrupt will be sent to the CPU. The CPU responds to this interrupt by:

1. Checking the CSR to determine the interrupt;
2. Checking the Tx DMA counter to disable the interrupt;
3. Restarting the system DMA for another Tx transfer; and
4. Setting the Tx DMA enable bit (COR1) to continue further Tx DMA requests by the UART.

The TX DMA counter 24 will be automatically reloaded after the Tx DMA enable bit is set. The CPU may reprogram the DMA counter 24 at anytime. If the DMA counter 24 is reprogrammed to 0, or if it is not programmed after a reset, then the external FIFO control is not active.

If the number of bytes to be transferred via Tx DMA is less than an integral number of bus widths, one of two options is available.

1. If the system DMA is sophisticated enough to recognize and signal the less-than-bus-wide transfer of data to the UART, then it can execute the transfer itself through /HBE and A0 inputs;
2. If the DMA can't execute this transfer, then the CPU must do the last transfer.

Each serial channel of the UART can also control internal and external FIFOs for the receiver. The internal Rx FIFO 12 for each channel is 16 bytes deep. External Rx FIFOs can be up to 16.777216 Mbytes deep.

In an Rx FIFO transfer without DMA, bits 2 and 1 in the ROR field are set to determine the number of bits in the Rx FIFO 12 before an interrupt is triggered (00 = 1 char., 01 = 1/4 full, 10 = 1/2 full, 11 = 7/8 full). Setting bit 7 in the IER field allows an interrupt to be issued when Rx FIFO 12 fills to the predetermined trigger level. Bits 4 and 3 of the ROR field determine the duration, expressed in character times, that a data byte must wait in Rx FIFO 12 until a timeout interrupt is issued to the CPU. This interrupt, if enabled, is issued if at least one data byte has been in Rx FIFO 12 for the number of character times specified and there has been no CPU access or serial data entering Rx FIFO 12 during that time. Setting bit IER11 enables timeout interrupts.

The CPU, upon receiving an interrupt will read the IIR field and check each bit for a pending interrupt. Finding bit 7 set indicates that Rx FIFO 12 is at its trigger level or a timeout interrupt is pending. The CPU reads the Bytes Till exception Register (BTE) 18 to get the count of the number of valid bytes it can remove from Rx FIFO 12. Then it reads these bytes from the DATA Register until that count is reached.

Since the CPU data bus can be wider than one byte and the serial data is always received in one byte increments, it is possible to have fewer bytes in Rx FIFO 12 than the bus is wide. This possibility will only occur during the last access by the CPU before the count of BTE 18 reaches zero. For example, if the data bus is 2 bytes wide and there are 15 bytes to be removed from Rx FIFO 12, as indicated by BTE 18, the CPU can do seven consecutive 2 byte wide reads without any further need to check the data status. Before the last read of the DATA Register, the CPU reads the FSR field to obtain the valid byte positions. It then reads the DATA Register for the last time to extract the remaining data bytes.

If another byte enters Rx FIFO 12 after the CPU has read the FSR field, but before it has read the DATA Register, the new byte will not be placed at the top of Rx FIFO 12. It is not added to the data that the CPU could read from Rx FIFO 12 until after the read of the DATA Register is finished.

DMA transfers using only Rx FIFO 12 are prepared for by executing the following initialization steps. The Rx FIFO trigger level bits in the ROR field (ROR3,2) and the timeout delay bits ROR5,4 are set to the required level. The Rx DMA mode select bit (COR2) is programmed for either single or multiple DMA transfers (0 and 1, respectively). Assuming DMA mode 1 is selected and Rx DMA enabled (COR 2.0 set), the following occurs.

Automatic DMA requests to transfer data from the UART will begin whenever the data reaches the trigger level. All consecutive valid data will be transferred until either an exception is encountered or until the number of valid data bytes left is less than one bus width. If the first exception encountered is an empty Rx FIFO 12, DMA requests will stop until the trigger level is reached.

If the first exception is a line status error, DMA transfers will stop when this byte is among those to be transferred next. The appropriate line status error bit(s) will be set in the IIR field and in BSR 16. If enabled, an interrupt will be issued to the CPU. At this point, the CPU responds as follows:

1. Read the CSR field to determine the type of interrupt(s);
2. Read the FSR field to determine the position of valid data;
3. Read BSR 16 to pinpoint the specific nature of the exception and clear the interrupt(s);
4. Execute the service routine for the specific LSI; and
5. Clear Rx FIFO 12 and execute the purge routine to eliminate the remaining incoming data associated with that block, then request retransmission of the data.

Automatic Rx DMA transfers will again start after the trigger level is reached or a timeout occurs.

If the first exception is a Comparison Register Match, the appropriate bits in the IIR field and BSR 16 will be set. If enabled, an interrupt will be issued to the CPU. At this point, the CPU responds as follows:

1. Read the CSR field to determine the type of interrupt(s);
2. Read the FSR field to determine the valid data byte locations;
3. Read BSR 16 to identify the matched bytes;
4. Read the Data Register, discarding the matched bytes and keeping the data bytes; and
5. Execute the appropriate control character service routine.

If the number of consecutive valid data bytes at the top of Rx FIFO 12 is less than one data bus width wide (e.g., 3, 2, or 1 byte in 32-bit access mode; 1 in 16-bit access mode), then DMA requests will stop and the UART will wait for enough bytes to arrive to reach the trigger level or until a timeout occurs. If a timeout occurs, the CPU will be required to remove the remaining valid data bytes and set the Rx DMA enable bit to restart the Rx DMA requests. Procedures for handling less than a data bus width of data are as follows:

1. Check the CSR field to determine the interrupt and consecutive valid data;
2. Read Rx FIFO 12 to extract all remaining bytes (check the FSR field when appropriate);
3. Prepare system DMA for next transfer; and
4. Set the Rx DMA enable bit to allow DMA transfers to continue.

DMA mode 0 operation is handled in the same way as mode 1, except that DMA requests are started as soon as one data bus width of valid data (4 bytes in 32 bit mode, 2 in 16 bit mode) has been accumulated. Automatic DMA request stops for any exceptions mentioned for DMA mode 1. DMA transfers will restart (according to mode 0 criterion) after any exceptions, timeouts or insufficient valid data width conditions have been cleared and the Rx DMA enable bit is set.

The UART also provides for maintaining an external Rx FIFO in conjunction with a system DMA unit. This is done to allow increased Rx FIFO length without significantly increasing the UART die size. Referring to Fig. 8, to do this, control circuitry that requests DMA transfers and notifies the CPU when a preselected number of bytes has been transferred is provided. One additional register is provided for external Rx FIFO control. This is the Rx CNT count register 26. It keeps track of the number of bytes transferred from Rx FIFO 12 via DMA. The CPU programs this counter with the number of bytes to be transferred. The counter decrements with each valid byte transferred. When it reaches 0, the Rx CNT bit in the IIR field is set and if enabled (IER7 set) an interrupt is issued. The Rx DMA bit in the control register (COR0) is cleared at this time, disabling further Rx DMA. UART operation reverts to the non-DMA transfer mode until the CPU sets the Rx DMA enable bit, again. The CPU responds to the Rx CNT interrupt by:

1. Reading the CSR field to determine the interrupt;
2. Reading the Rx CNT register the clear the interrupt;
3. Processing the bytes stored in the external Rx FIFO;
4. Preparing the system DMA for subsequent transfers; and
5. Setting the Rx DMA enable bit in the Channel Operation register to allow subsequent DMA transfers.

In mode 1, DMA reaching trigger level of Rx FIFO 12 starts the transfer to the external Rx FIFO. The transfer continues until all consecutive valid data bytes in Rx FIFO 12 have been transferred, there is a line

status condition or until a byte match condition occurs at the top of Rx FIFO 12. If the counter is not zero and there is less than one bus width of consecutive valid data at the top of Rx FIFO 12, then the UART will wait for enough bytes to arrive to reach the trigger level and continue a full bus width DMA transfer. If enough bytes don't, arrive a timeout interrupt will be issued to the CPU. The CPU response to a timeout interrupt is:

1. Check the CSR field, to determine the interrupt and consecutive valid data;
2. Check the RCR to clear the interrupt and determine the number of bytes in the external Rx FIFO that need to be processed;
3. Read Rx FIFO 12 to extract all remaining bytes (check the FSR field when appropriate);
4. Process the bytes stored in the external Rx FIFO; and
5. Start a routine to handle the remaining incoming bytes in this block.

The Rx DMA counter 26 is reloaded to the preprogrammed value after the Rx DMA enable bit is set. The enabled Rx DMA transfer will start again, as soon as, the trigger level is reached. In DMA mode 0, transfer will start, as soon as enough data is assembled to make one complete bus width transfer.

- 15 The CPU may reprogram the DMA counter at anytime. If the DMA counter is reprogrammed to 0 or it is not programmed after a chip reset, then external Rx FIFO control is not active. Line status or byte match conditions are handled the same way as in the internal Rx FIFO mode.

There are eight types of interrupts the CPU can enable via the IER field. Two report receiver status, three report transmitter status and the other three report input status, line status and comparison register matches.

Setting bit IER7 allows an Rx FIFO 12 at trigger level indication (IIR7) to activate the interrupt. If this interrupt is pending, the CPU response is:

1. Check the IIR field to determine the interrupt;
2. Check the Bytes Till Exception Register 4 to determine the number of bytes till an exception;
- 25 3. Extract all of the consecutive valid data bytes till an exception (bit IIR7 is cleared when the data falls below the trigger levels); and
4. Read the FSR and BSR fields and extract any remaining valid data bytes along with any pertinent line status or match information about the invalid data bytes.

Setting bit IER7 when the external Rx FIFOs are in use allows the RCR (IIR1) indication to activate the interrupt signal. The RCR indication IIR1 is set when the RCR reaches 0.

Another Rx interrupt enabled by setting bit IER1 is the timeout interrupt. This indication goes active when an internal timer reaches zero. The indication means that data has been sitting in Rx FIFO 12 longer than a preprogrammed limit without the CPU taking action or additional data arriving. This timeout interrupt prevents data that can't reach the trigger level from being stuck in the FIFO indefinitely. It triggers the same indication (IIR7) as RFT, because the CPU response is the same for timeout, as for the RFT indication.

The internal timer for timeout indication is started when the first byte enters Rx FIFO 12 and is stopped when the last byte is removed from Rx FIFO 12. The timer is automatically restarted each time the CPU removes a byte from Rx FIFO 12 or each time the Rx shift register 27 loads a byte 27 from Rx FIFO 12 or each time Rx shift register loads a byte into it. The duration of the timer is programmed by setting bits 4 and 3 in the ROR. All timer durations are increments of 1 character time (00 = 1 char., 01 = 2 chars., 1 = 3 chars., 11 = 4 chars.). If the programmed duration of the timer expires before the CPU accesses Rx FIFO 12 or another retrieved byte is loaded into Rx FIFO 12, then the timeout indication (IIR7) is set.

An external Rx FIFO is considered to be in use when both the Rx DMA enable bit is set and the RCR 26 is loaded. The Rx timeout operation when using an external Rx FIFO is identical to that listed above for Rx FIFO 12, except in two respects.

1. The timeout duration is lengthened by 100x. This is done because it is assumed that the system will be receiving files instead of accepting input from a user terminal and, therefore, the timeout durations should be much longer. Specifically, if the file is being transferred at 9600 baud using 8 data, 1 stop and no parity, then the Rx timeout delays available to the user are approximately 100, 200 300, 400 ms.
- 50 2. Once started, the timeout timer is only, disabled if both Rx FIFO 12 is empty and RCR 26 = 0. This allows the timer to monitor time delays for both the internal and external Rx FIFOs. This is useful because data could conceivably be stuck in either internal, external or both Rx FIFOs.

Setting bit IER6 allows the Comparison Register Match indication to issue an interrupt. The comparison match will occur when the Comparison Register enable bit TOR6 is set and a data byte at the top of Rx FIFO 12 matches a byte loaded into a Comparison Register. In response to the interrupt, the CPU will:

- 55 1. Read the IIR field to determine the interrupt;
2. Read the FSR field to determine if there is valid data at the top of Rx FIFO 12;
3. Read BSR 16 to determine the byte that matched and what it was (this clears the interrupt); and

4. Read the DATA Register.

Setting bit IER5 allows Line Status Indications (LSI) to activate an interrupt when the bytes come to the top of Rx FIFO 12. These occur due to either an overrun error (OE), parity error (PE), framing error (FE) or a break indication (BI). The specific indication is associated with the byte that caused it in BSR 16. However, the OE will be indicated as soon as the UART recognizes this condition. In response to an LSI, the CPU will:

1. Check the IIR field to determine the interrupt;
2. Check the FSR field to determine if there is valid data at the top of Rx FIFO 12;
3. Check BSR 16 to determine the byte(s) that caused the LSI (this clears the interrupt); and
4. Read Data Register 10.

Setting bit IER4 allows the Input Status Indication (ISI) to activate an interrupt. An ISI occurs when any of the general-purpose input lines change. If bit IER4 is set then this indication will activate an interrupt. The CPU response to this interrupt is:

1. Read the IIR field to determine the interrupt; and
2. Read the DIR field (this clears the interrupt).

Setting bit IER3 allows the Tx flow control indication (TFC) to activate an interrupt. The TFC indication occurs when a change to the Tx enable bit has been made due to a match in Rx FIFO 12 with cComparison Register 0 or 1 or due to a change on the CTS pin. This assumes that these flow control options are enabled. The Tx flow control consists of automatically starting or stopping the transmitter when user determined bytes such as XON or XOFF are detected in the data stream. Reading the TOR field to check the Tx enable state clears the interrupt. Its purpose is to inform the CPU that a change of state has occurred to the Tx enable bit. The CPU can override an automatic setting of the Tx enable bit at anytime by simply writing to it. This interrupt occurs immediately upon a match if TOR3 is set. If TOR3 is 0 and a match occurs, then the interrupt goes active when the flow control character reaches the top of the FIFO. If TOR3 is set, then the flow control characters are not put into the FIFO.

Setting bit IER2 allows the Tx FIFO empty indication or the Tx DMA count = 0 indication to activate an interrupt. When Tx FIFO 14 is empty, an indication is given through bit IIR2. The CPU responds to this interrupt by:

1. Reading the IIR field to determine the interrupt; and
2. Either loading data into Tx FIFO 14 or if it's the start of Tx DMA, enabling the Tx DMA bit (this clears the interrupt). If there is no data to transmit the CPU can clear the Tx enable bit to clear the interrupt. If an external Tx FIFO is in use, then the IIR2 indication is activated when the Tx DMA counter reaches its 0 count.

Once the Comparison Registers have been programmed and enabled, each byte entering Rx FIFO 12 is checked for a match. The UART contains 8 Comparison Registers and only those that are explicitly loaded by the CPU are actively compared. Therefore, any number of "control" characters up to 8 can be in use. A match with any Comparison Register, except Comparison Registers 0 and 1, will be indicated by bit IIR6 and the appropriate BSR bits when it is among the next data to be taken from the top of Rx FIFO 12. Assuming an active interrupt was caused by a Comparison Register match, the CPU will:

1. Read the IIR field to determine the interrupt;
2. Read the FSR field to determine the valid data at the top of Rx FIFO 12;
3. Read BSR 16 to determine which byte matched and what it was; and
4. Read Data Register 10 to extract the bytes from the Rx FIFO 12. The CPU retrieves the valid data sorting it from the control character(s). A match with the Comparison Registers handling flow control (0 and 1) is indicated by IIR3.

Transferring byte-wide serial data to a word-wide (I.E. 16-BIT) parallel data bus and vice versa requires an additional constraint that is not present memory/CPU exchanges. In theory, the CPU may access each serial channel any one of three ways: odd byte, given byte or word transfers. Because the data from each channel is always ordered bytes-by-byte, it must be presented to the CPU with the original order preserved. Two signals issued by the CPU typically control the way in which the data will be transferred. The /HBE and A0 input signals to the UART, which are used to control byte transfers indicate the number of bytes and which part of the 16-bit CPU bus is used, not which bytes in the UART FIFOs will be accessed.

The following describes typical 16-bit bus transfers to a peripheral and then the 16-bit bus transfers to the UART.

As stated above, a 16-bit wide data bus allows the CPU to transfer data to or from a peripheral device in three different ways, Table 1 below sets forth the manner in which the UART will enable the data bus for the given states of /HBE and A0. During conventional transfers, the CPU establishes the control signals /HBE and A0 and transfers the data utilizing normal transfer procedures. The data, if it is being sent, has a

predetermined location to go to based upon the address and the data position on the data bus. This is true for each of the three types of transfers. If the data is being received by the CPU, then it is also handled in the CPU based upon the address and the data's position on the data bus.

When simultaneously transferring from the CPU to the UART 16 bits of data that is to be transmitted serially, the data's predetermined position in the UART FIFO depends upon the address and the data bytes' position on the data bus. However if only 8 bits of data is being sent to the UART for serial transmission, then the predetermined position in the UART's FIFO depends upon the address and the number of bytes being sent, not by the byte's position on the data bus.

When the transferred data has been received serially by the UART and will be transferred across a 16-bit bus to the CPU, the data's "order of reception" must be preserved during the 16-bit data bus transfer. During a 16-bit wide data transfer, this order is preserved with conventional constraints of address and data bus position. However, during an 8-bit transfer the next byte available in the UART FIFO is transferred to the data bus regardless of the position on that bus that it will occupy, in other words, regardless of whether it will be in the high byte or low byte position.

TABLE 1

TRANSFER TYPE	/HBE	A0	BUS ACTIVITY
even word	0	0	D15-D0
odd byte	0	1	D15-D8
even byte	1	0	D7-D0

As stated above, the odd and even byte transfers determined on which part of the bus the next byte in the Rx FIFO 12 will be placed. The Data and Status Registers operate in the same manner. Status must be read before data in order to associate the status byte with the appropriate data byte.

Writing to the DATA Register loads data into Tx FIFO 14. Data can be written to the DATA Register using any /HBE and A0 combination. During single byte transfers, the data byte will be located in the next available space in Tx FIFO 14, regardless of its position on the CPU bus (D15-D8 or D7-D0). Word wide data is loaded into Tx FIFO 14 assuming that the data byte on D7-D0 should be transmitted immediately before the data on D15-D8.

All other registers besides Data and Status can be addressed on a byte-by-byte basis using /HBE and A0. For example, in the 16-bit access mode, if the register at address 5 is accessed with /HBE=0 and A0=1, then the register contents will be placed on D15-D8. Writes to odd and even byte wide registers are accomplished in a similar way using these strobes, as are word wide strobes.

It should be understood that various alternatives to the embodiment described herein may be employed in practicing the present invention. It is intended that the following claims define the invention and that methods and structures within the scope of these claims and their equivalents be covered thereby.

Claims

1. In a data communications device that transfers digital data characters received from a first station to a second station and that includes receiver memory means having storage capacity for a plurality of received data characters, the improvement comprising a counter connected to the receiver memory means for indicating a number of consecutive valid data characters that have been received by the receiver memory means for transfer to the second station before an exception is encountered.
2. A data communications device as in claim 1 and further including means responsive to a read access request from the second station for transferring the number of consecutively received valid data characters to the second station.
3. A data communications system comprising
 - (a) a first station;
 - (b) a second station that includes means for issuing a read access request; and
 - (c) a data communications device that transfers data characters received from the first station to a second station, the data communications device comprising
 - (i) receiver memory means having a storage capacity for a plurality of data characters received from the

first station;

(ii) means responsive to the receiver memory means for indicating the number of consecutively received valid data characters remaining in the receiver memory means for transfer to the second station upon receipt of a read access request from the second station;

5 and

means responsive to a read access request from the second station for transferring the number of consecutively received valid data characters to the second station.

4. A data communications device that performs serial-to-parallel conversion on digital data characters received from a first station for transfer to a second station, the data communications device comprising

10 (a) a receiver shift register that receives data characters serially transferred from the first station;

(b) receiver memory means for storing a plurality of data characters received from the receiver shift register for transfer to the second station;

(c) means for identifying whether each data character received by the receiver memory means will trigger an exception;

15 (d) means for indicating a number of consecutive data characters stored in the receiver memory means that are available for transfer to the second station until an exception is triggered; and

(e) means for transferring the number of consecutive data characters to the second station.

5. A data communications device as in claim 4 wherein the second station comprises a central processing unit, data storage means associated with the central processing unit, and means for transferring the consecutive valid data characters directly from the data communications device to the data storage means.

6. A data communications device as in claim 4 wherein the means for identifying includes means for comparing the data characters received from the first station with preselected comparison information stored in the means for comparing.

25 7. A data communications device as in claim 6 and further including means for changing the preselected comparison information.

8. A data communications device as in claim 4 wherein the means for identifying includes means for identifying line status errors in data characters received from the first station.

9. A data communications device as in claim 4 wherein the means for identifying valid data characters includes means identifying that a data character received by the data communications device is the most recent data character received by the data communications device.

10. A data communications system comprising:

(a) means for storing a plurality of data characters received by a data communications from a first station;

35 (b) means for determining whether each received data character will trigger an exception;

(c) means for maintaining a count of the number of consecutive stored data characters that can be transferred to a second station before an exception is encountered;

(d) means for transferring the number of consecutive stored data characters to the second station.

40 11. A method of reading a plurality of data characters from a data communications device that receives data character from a first station and transfers the received data characters to a second station, the method comprising:

(a) storing a plurality of data characters received by the data communications device from the first station;

(b) determining whether each received data character will trigger an exception;

45 (c) maintaining a count of the number of consecutive stored data characters that can be transferred to the second station before an exception is encountered;

(d) transforming the number of consecutive valid data characters to the second station.

50 12. A method as in claim 11 wherein the second station comprises a central processing unit, storage means associated with the central processing unit, the method includes the step of transferring the consecutive valid data characters directly to the storage means.

13. A method as in claim 11 wherein the step of determining whether each received data character will trigger an exception includes the step of comparing each received data character with preselected comparison information such that a match between the received data character and the preselected comparison information results in that data character have been determined as a data character that will

55 trigger an exception.

14. A method as in claim 1 step of changing the preselected comparison information.

15. A method as in claim 11 wherein the step of determining whether each received data character will trigger an exception includes the step of determining whether a received data character includes a-line

status error wherein transfer to the second station of a data character including a line status will trigger an exception.

16. A method as in claim 11 wherein the step of determining whether each received data character will trigger an exception includes the step of determining whether a received data character is the most recently stored data character wherein transfer to the second station of the most recently stored data character will trigger an exception.

17. In a data communications device that includes a plurality of communications channels for transferring data characters between a first system and a plurality of peripheral systems, each of the channels being associated with a corresponding peripheral system for data character transfer between that peripheral system, and the first system, wherein each channel includes an identical register set having for storing digital information the improvement comprising an alternate function selector for enabling concurrent writes to the same selected register in each channel's register set.

18. A data communications device that transfers data characters between a plurality of peripheral stations and a common central station, the data communications device comprising

(a) a plurality of communications channels, each channel being associated with a corresponding one of the peripheral stations for data character transfer between that peripheral station and the central station, each channel including an identical register set that includes at least one register for storing digital information relating to the data character transfers;

(b) means for loading digital information into the register sets; and

(c) means connected to each of the channels for enabling concurrent writes to the same selected register in each channel's register set.

19. A data communications device as in claim 18 wherein the means for enabling comprises an alternate function register in each of the channel's register sets, each alternate function register including a bit which if enabled in any alternate function register enables concurrent writes to the same selected register in each channel's register set.

20. A method of transferring data characters from the system memory of a data processing system having a direct memory access unit associated therewith to a data communications device utilizing the direct memory access unit, the data communications device including transmitter memory means having storage locations for a plurality of data characters received by the data communications device from the system memory, the method comprising:

(a) determining whether the transmitter memory means is empty;

(b) establishing a count of the number of data characters that are to be transferred from the system memory to the data communications device;

(c) enabling the direct memory access unit for direct transfer of the number of data characters from the system memory to the data communications device; and

(d) transferring the number of data characters to the data communications device utilizing the direct memory access unit.

21. A data processing system comprising:

(a) a data processing system having a direct memory access unit associated therewith for transferring data characters from the system memory of the data processing system to a data communications device utilizing the direct memory access unit;

(b) a data communications device comprising

(i) transmitter memory means having storage locations for a plurality of data characters received by the data communications device from the system memory;

(ii) means for determining whether the transmitter memory means is empty;

(iii) means for establishing a count of the number of data characters that are to be transferred from the system memory to the data communications device utilizing the direct memory access unit;

(iv) means for enabling the direct memory access unit for direct transfer of the number of data characters from the system memory to the data communications device; and

(v) means for transferring the number of data characters to the data communications device utilizing the direct memory access unit.

22. A method of transferring data characters from the system memory of a data processing system having a direct memory access unit associated therewith to an external storage unit have storage locations for a plurality of data characters, the method comprising:

(a) determining whether the external storage unit is empty;

(b) establishing a count of the number of data characters that are to be transferred from the system memory to the external storage unit;

(c) enabling the direct memory access unit for direct transfer of the number of data characters from

the system memory to the external storage unit.

23. A method as in claim 23 and further including the steps of issuing an interrupt to the data processing system when the number of data characters have been transferred from the system memory to the external storage unit;

identifying the interrupt as indicating that the number of data characters has been transferred to the external storage unit;

disabling the interrupt;

reenabling the direct memory access unit to facilitate subsequent data character transfers;

24. A method of transferring data characters from a data communications device to the system memory of a data processing system utilizing a direct memory access unit associated with the data processing system, the data communications device including receiver memory means having storage locations for a plurality of data characters received by the data communications device from a peripheral system for transfer to the system memory, the method comprising

(a) maintaining a count of the number of data characters available for transfer from the receiver memory means before an exception is encountered;

(b) establishing a trigger level for the receiver memory means;

(c) initiating the transfer of data characters from the receiver memory means to the system memory utilizing the direct memory access unit when the receiver memory means trigger level is reached;

(d) continuing the transfer of data characters from the receiver memory means to the system memory utilizing the direct memory access unit until either an exception is encountered or until the number of valid data characters remaining to be transferred is less than the width of the data bus of the data processing system.

25. A method as in claim 25 wherein transfer of data characters continues until an exception is encountered, the exception being an empty receiver memory means, the method including the additional step of terminating all data character transfers from the receiver memory means until the trigger level of the receiver memory means is reached.

26. A method as in claim 25 wherein transfer of data characters continues until an exception is encountered, the exception being a line status error, the method including the following additional steps:

(a) issuing an interrupt to the data processing system;

(b) determining the type of interrupt;

(c) determining the location of valid data in the receiver memory means;

(d) determining the specific nature of the exception;

(e) clearing the interrupt;

(f) executing a service routine for the specific line status error interrupt;

(g) clearing the receiver memory means;

(h) executing a purge routine to eliminate the remaining incoming data associated with the data character that triggered the exception; and

(i) requesting retransmissions of the data associated with the data character that triggered the exception.

27. A method as in claim 25 wherein transfer of data characters continues until an exception is encountered, the exception being a match between a data character received by the data communications device and preselected comparison information, the method including the following additional steps:

(a) issuing an interrupt to the data processing system;

(b) determining the type of interrupt;

(c) determining the location of valid data characters in the receiver memory means;

(d) identifying the matched data character;

(e) reaching the first storage location in the receiver memory means, discarding the matched data character; and

(f) executing an appropriate control character service routine.

28. A method as in claim 25 wherein transfer of data characters continues until the number of data characters remaining to be transferred is less than the width of the data bus, the method including the additional step of terminating transfers from the receiver memory means utilizing the direct memory access unit until the trigger level has been reached.

29. A method as in claim 25 wherein transfer of data characters continues until the number of data characters remaining to be transferred is less than the width of the data bus and data character transfers have been discontinued for a preselected timeout period, the method including the following additional steps:

(a) issuing an interrupt to the data processing system;

(b) identifying the interrupt and the number of valid data characters remaining to be transferred;
 (c) transferring the number of consecutive valid data characters remaining to be transferred;
 (d) enabling the direct memory access unit for subsequent data character transfers from the receiver memory means to the system memory utilizing the direct memory access unit.

5 30. A data processing system for transferring data characters from a data communications device to the system memory of a data processing system utilizing a direct memory access unit associated with the data processing system, the data communication device including receiver memory means having storage locations for a plurality of data characters received by the data communications device from a peripheral system for transfer to the system memory, the data processing system comprising:

10 (a) means for maintaining account of the number of data characters available for transfer from the receiver memory means before an exception is encountered;

(b) means for establishing a trigger level for the receiver memory means;

(c) means for initiating the transfer data characters from the receiver memory means to the system memory utilizing the direct memory access unit when the receiver memory means trigger level is reached;
 15 and

(d) means for continuing the transfer of data characters from the receiver memory means to the system memory utilizing the direct memory access unit until either an exception is encountered or until the number of valid data characters remain to be transferred is less than the width of the data bus of the data processing system.

20 31. A method of transferring data character from a data communications device to an external storage unit for transfer to the system memory of a data processing system utilizing the direct memory access unit associated with the data processing system the external storage unit having storage locations for a plurality of data characters received from the data communications device the data communications device having storage locations for a plurality of data characters received by the data communications device from a peripheral system, the method comprising:

25 (a) establishing a preselected number of data characters that may be transferred from the receiver memory means to the external storage unit utilizing the direct memory access unit;

(b) establishing a trigger level for the receiver memory means;

(c) initiating the transfer of data characters from the receiver memory means to the external storage unit
 30 utilizing the direct memory access unit when the receiver memory means trigger level is reached;
 continuing the transfer of data characters from the receiver memory means to the external storage unit utilizing the direct memory access unit until either the preselected number of data character has been transferred, or an exception is encountered or until the number of valid data characters remaining to be transferred is less than the width of the data bus of the data processing system.

35 32. A method as in claim 32 wherein transfer of data characters continues until the receiver memory means trigger level is reached, the method including the additional steps of

(a) issuing an interrupt to the data processing system;

(b) reading the number of characters that have been transferred to the external storage unit;

(c) clearing the interrupt;

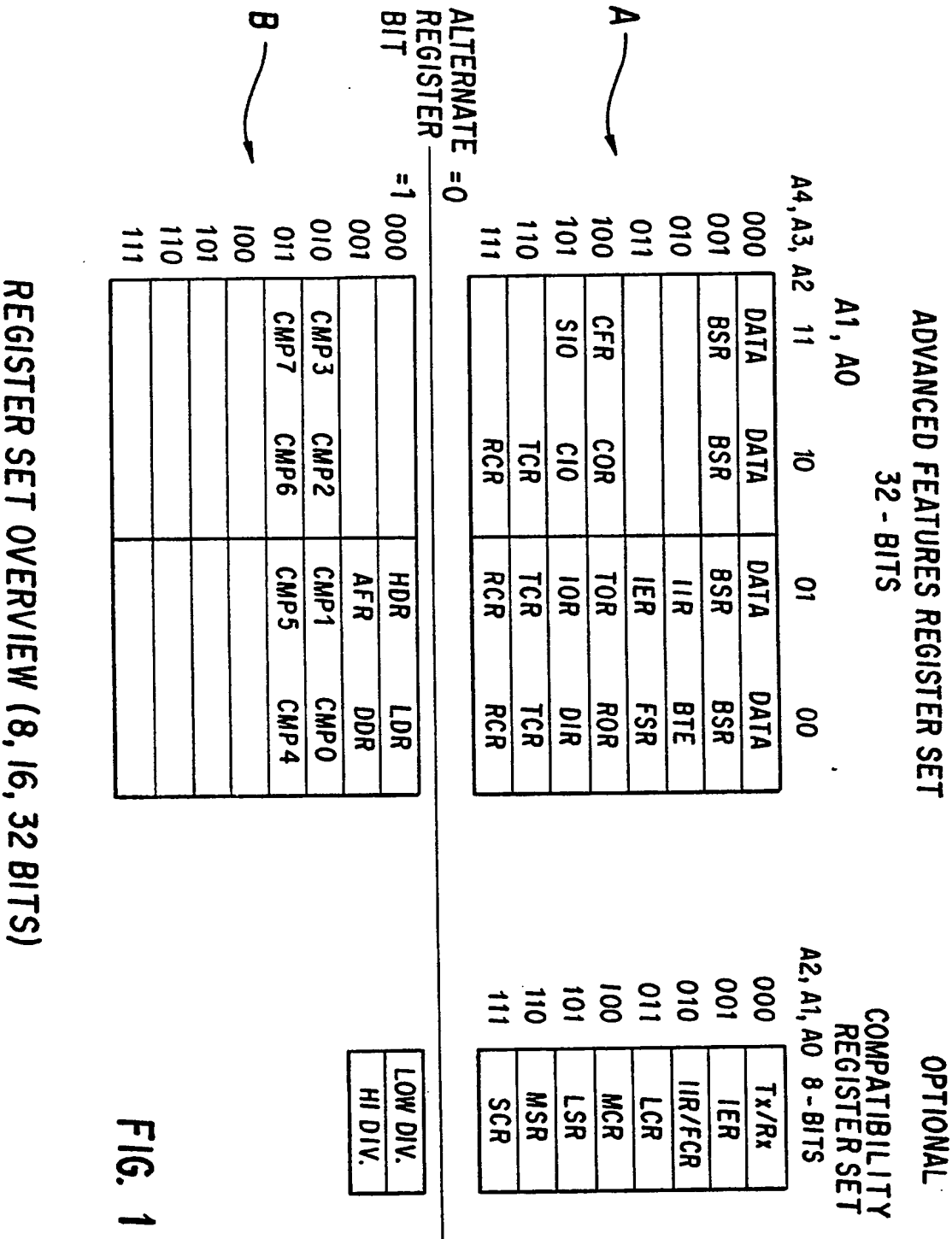
40 (d) processing the data characters stored in the external storage unit;

(e) enabling the direct memory access unit for subsequent data character transfers from the receiver memory means.

45

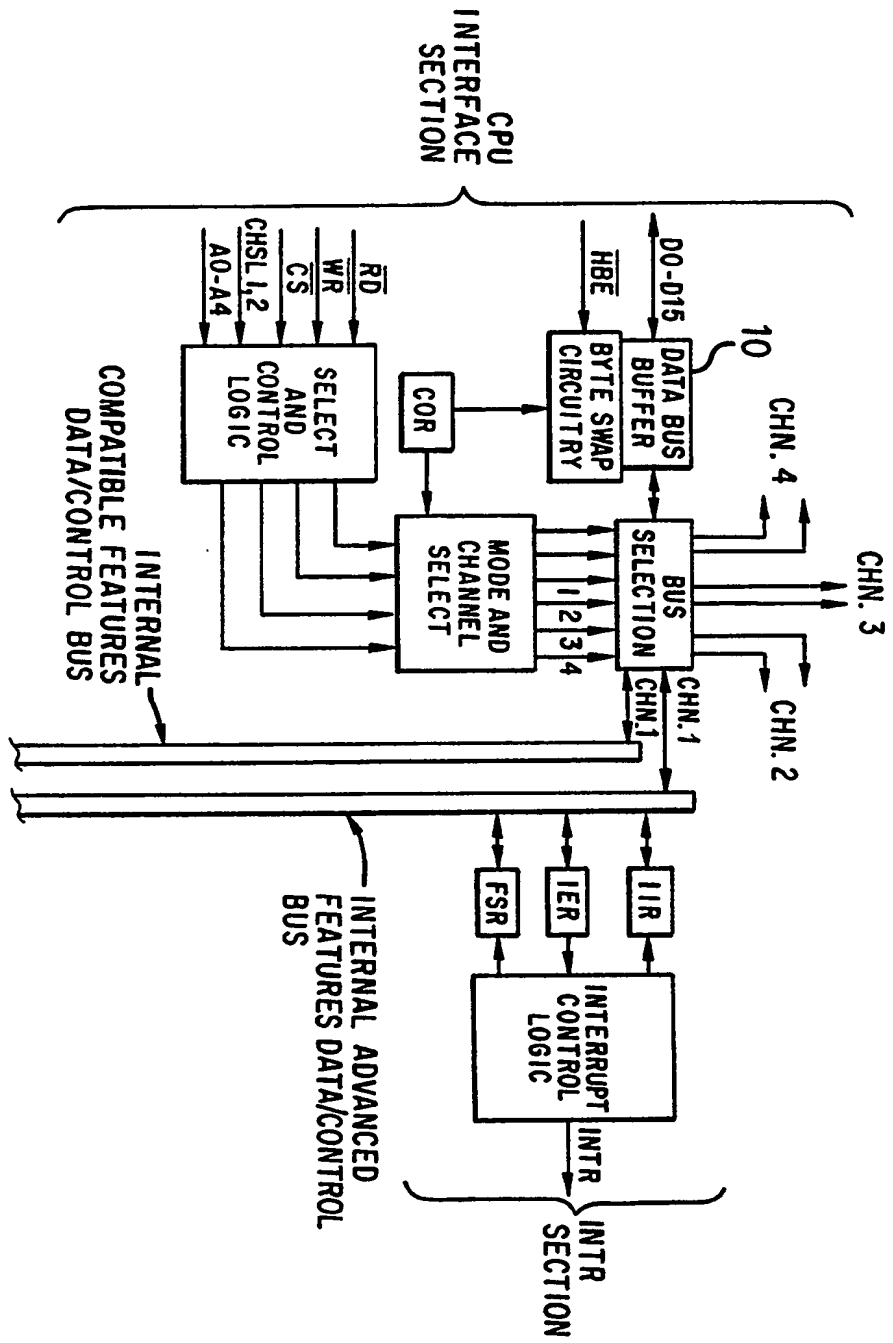
50

55



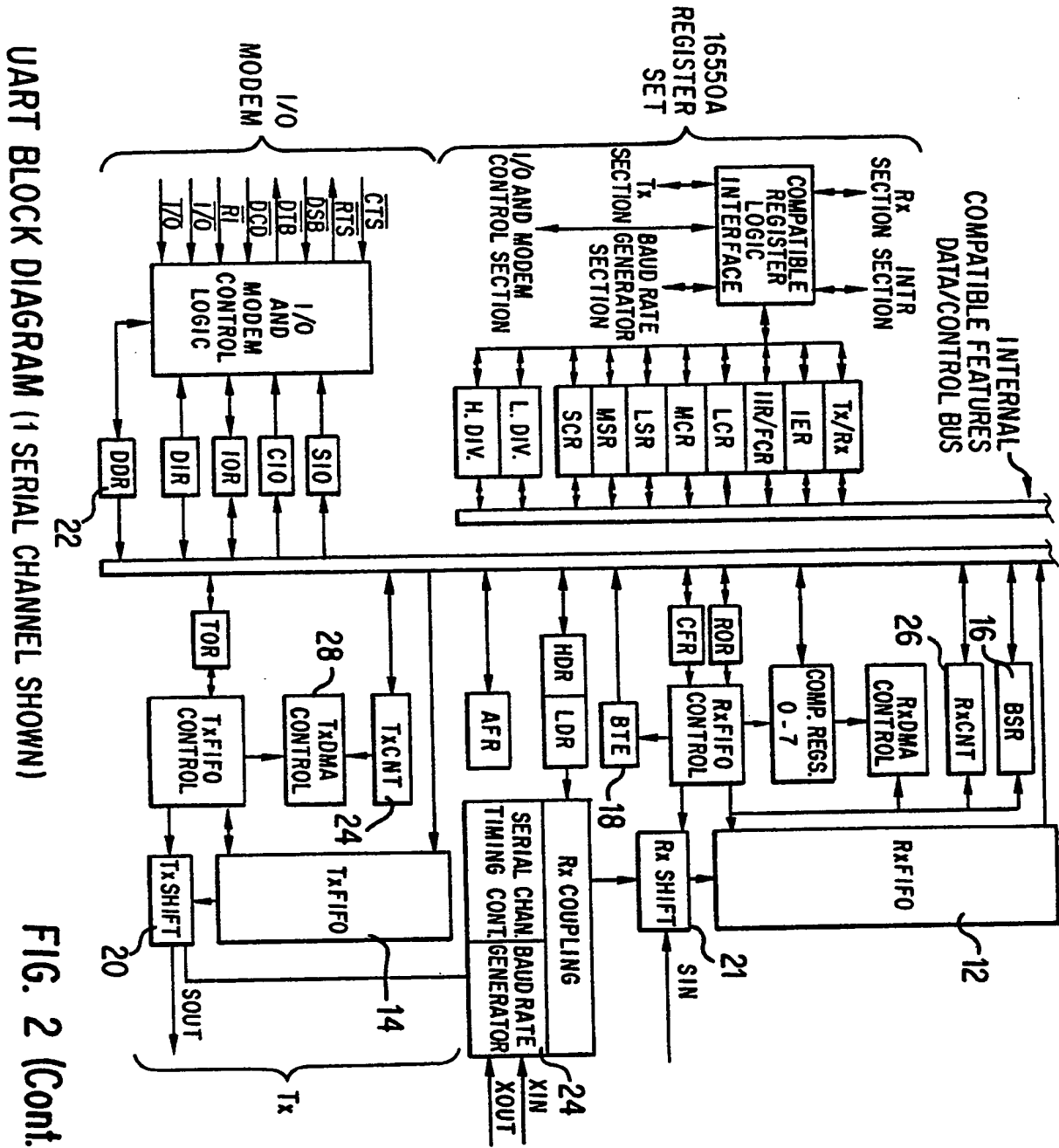
REGISTER SET OVERVIEW (8, 16, 32 BITS)

FIG. 1



UART BLOCK DIAGRAM (1 SERIAL CHANNEL SHOWN)

FIG. 2



UART BLOCK DIAGRAM (1 SERIAL CHANNEL SHOWN)

FIG. 2 (Cont.)

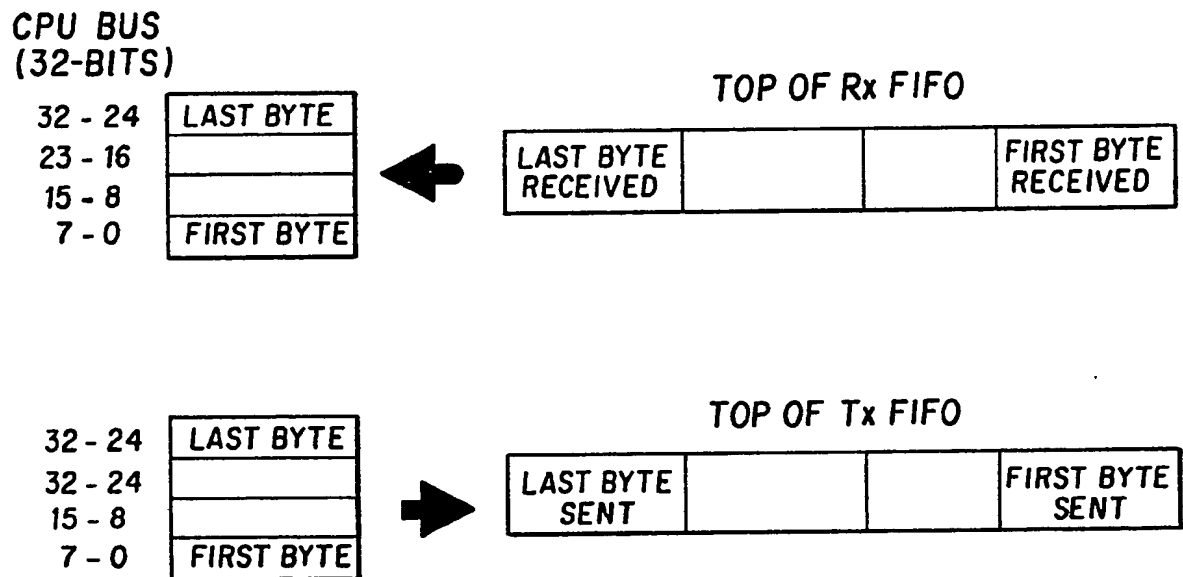
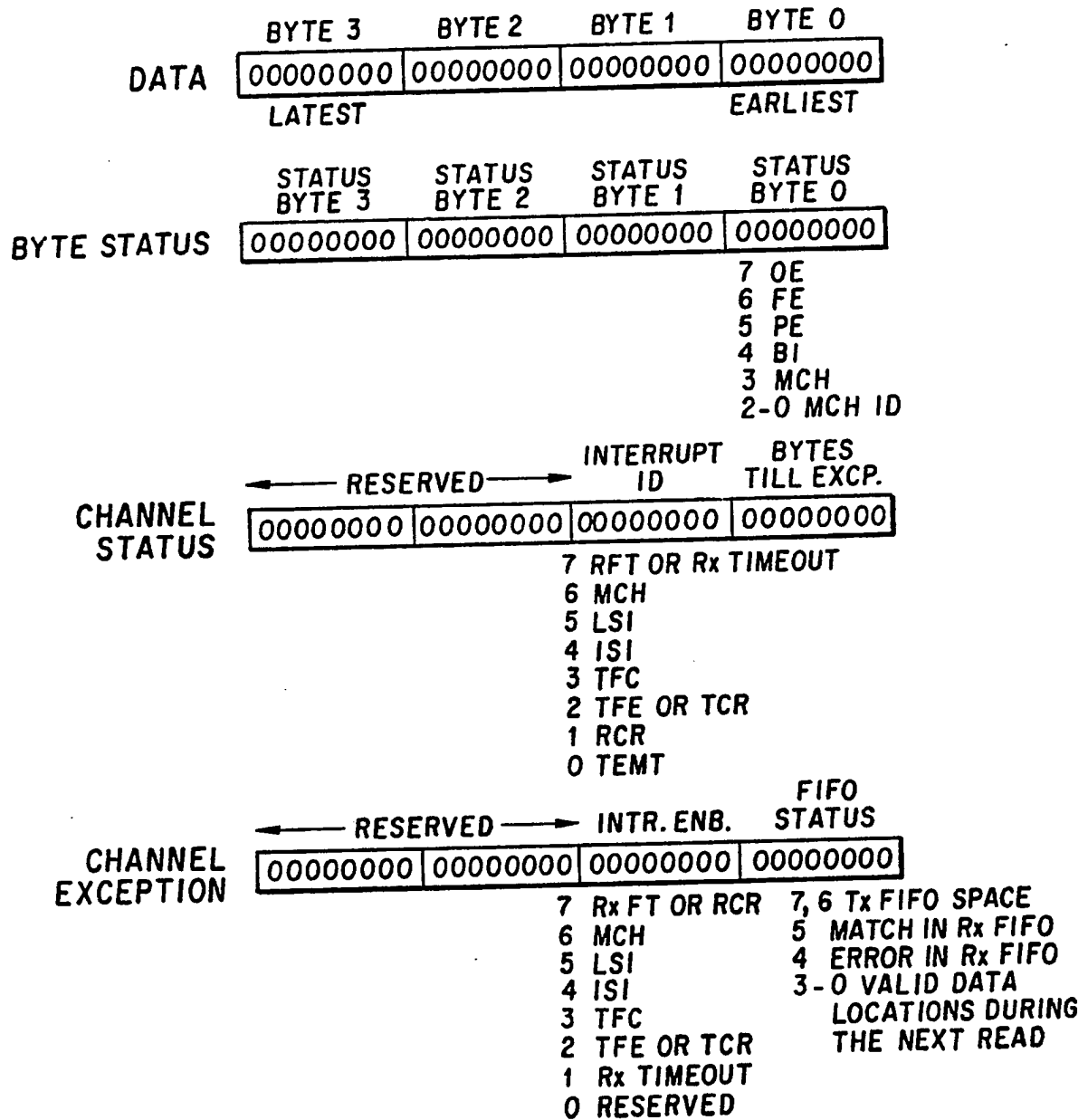
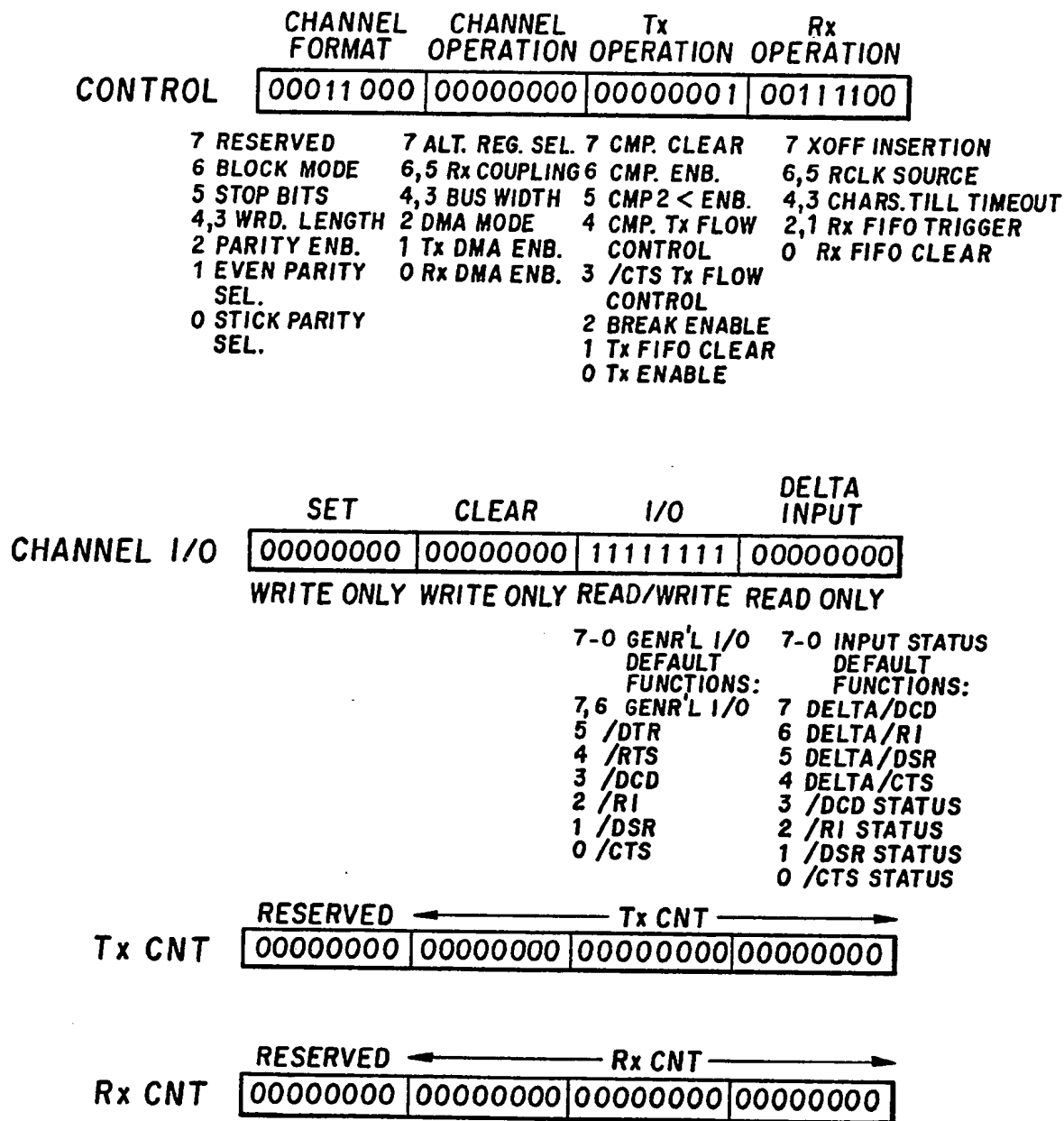


FIG. 3



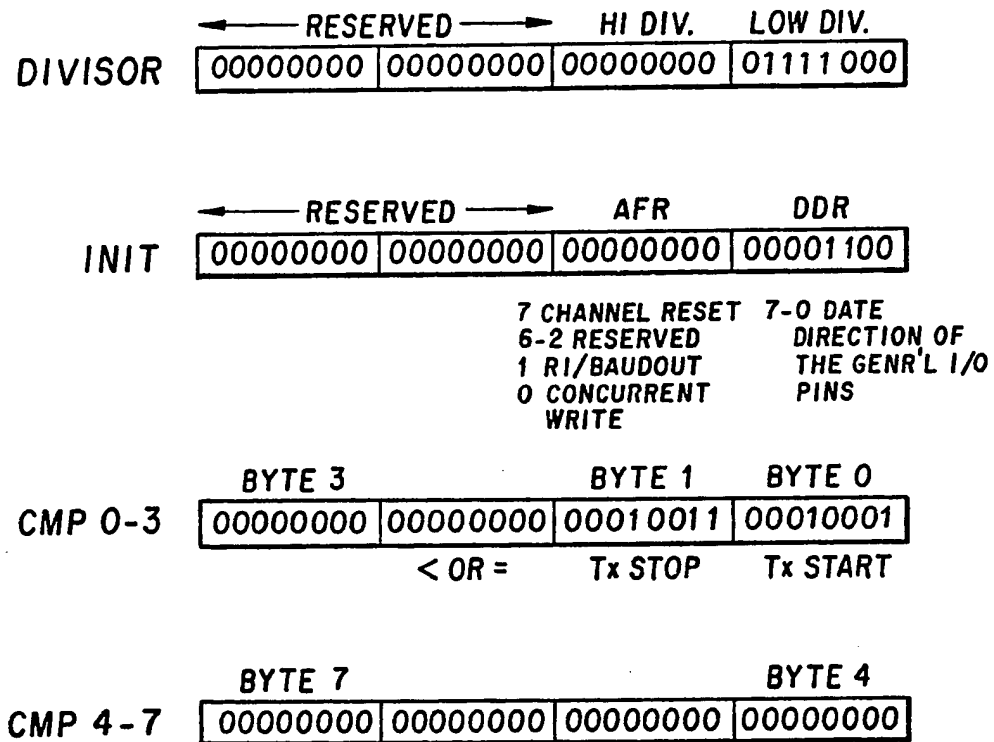
ADVANCED FEATURES REGISTER SET
(BITS INDICATE THEIR STATE AFTER RESET)

FIG. 4



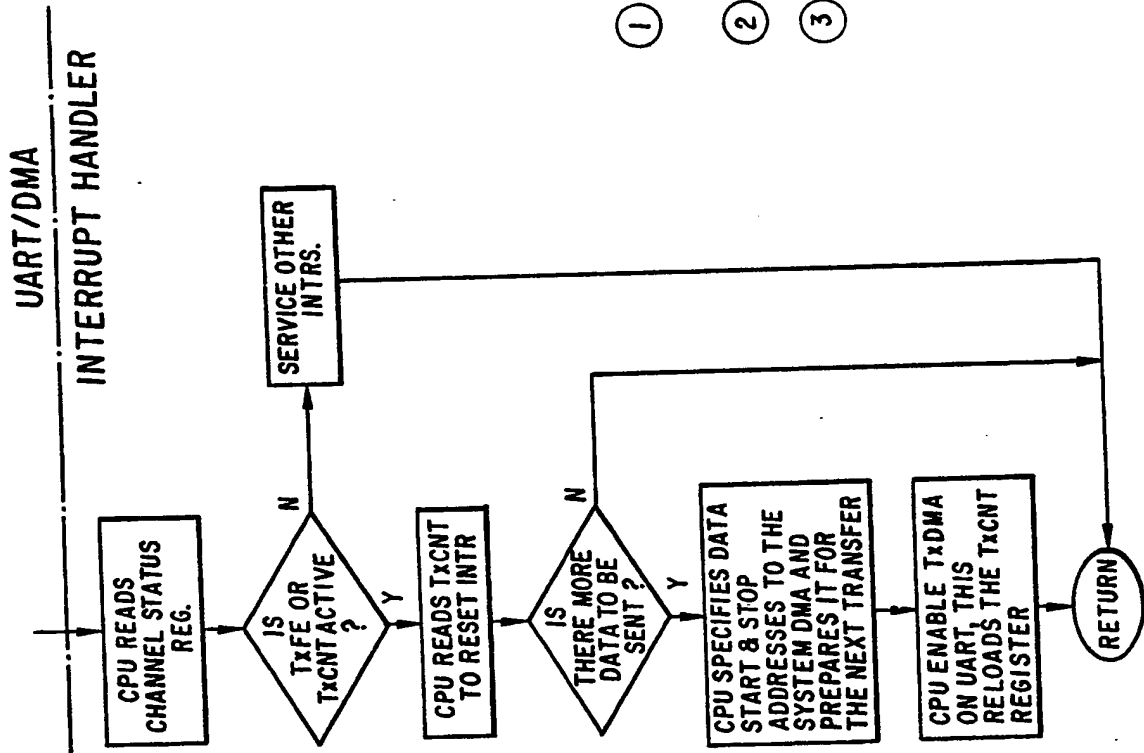
ADVANCED FEATURES REGISTER SET
(REMAINING REGISTERS; BITS
INDICATE THEIR STATE AFTER RESET)

FIG. 5



ALTERNATE REGISTER SET DETAIL REGISTER
(BITS INDICATE THEIR STATE AFTER RESET)

FIG. 6



① FOR TxDMA w INTERNAL TxFIFO ONLY, THE TxFE OR TxCNT INTERRUPT SHOULD NOT BE ENABLED.

② FOR TxDMA w EXTERNAL TxFIFO, THE TxFE OR TxCNT INTERRUPT SHOULD BE ENABLED

③ WHEN USING AN EXTERNAL TxFIFO ITS LENGTH SHOULD BE AN INTEGRAL MULTIPLE OF THE BYTES THAT ARE TRANSFERRED DURING A SINGLE DMA CYCLE (ie. A MULTIPLE OF 2 IF THE DATA BUS IS 2 BYTES WIDE).

Tx DMA PROCEDURE

FIG. 7-(Cont.)

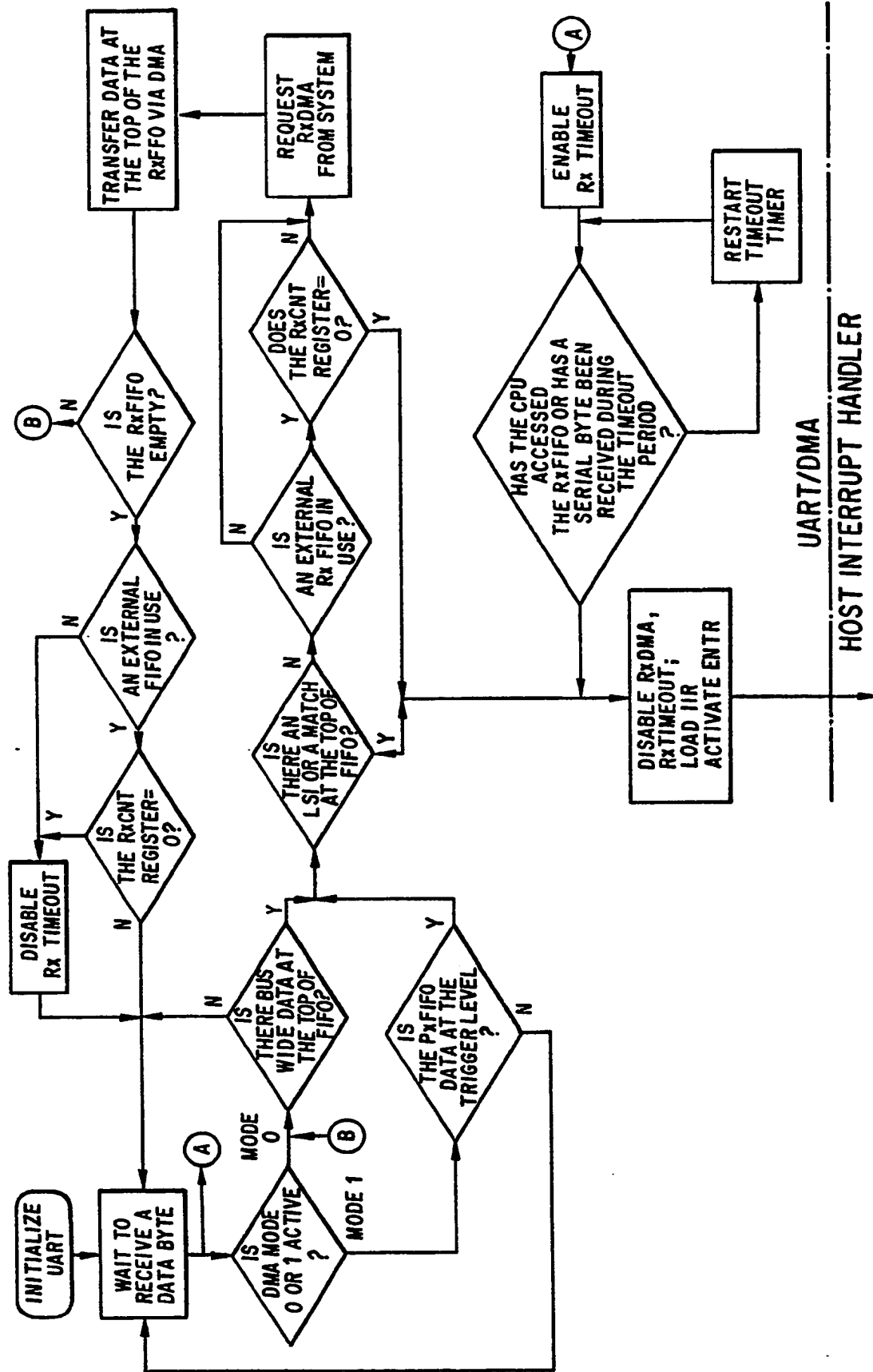
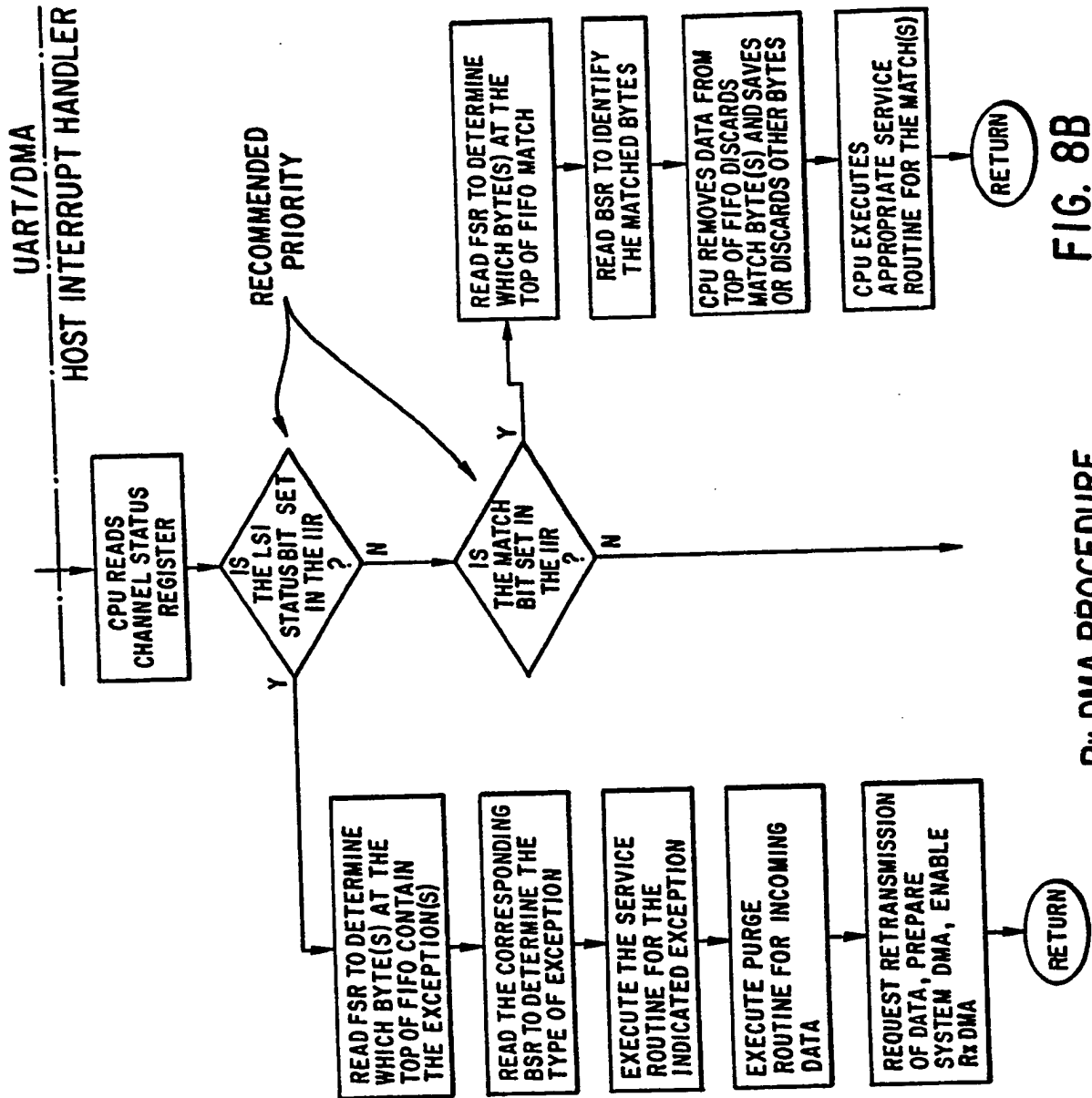


FIG. 8A

Rx DMA PROCEDURE

UART/DMA
HOST INTERRUPT HANDLER



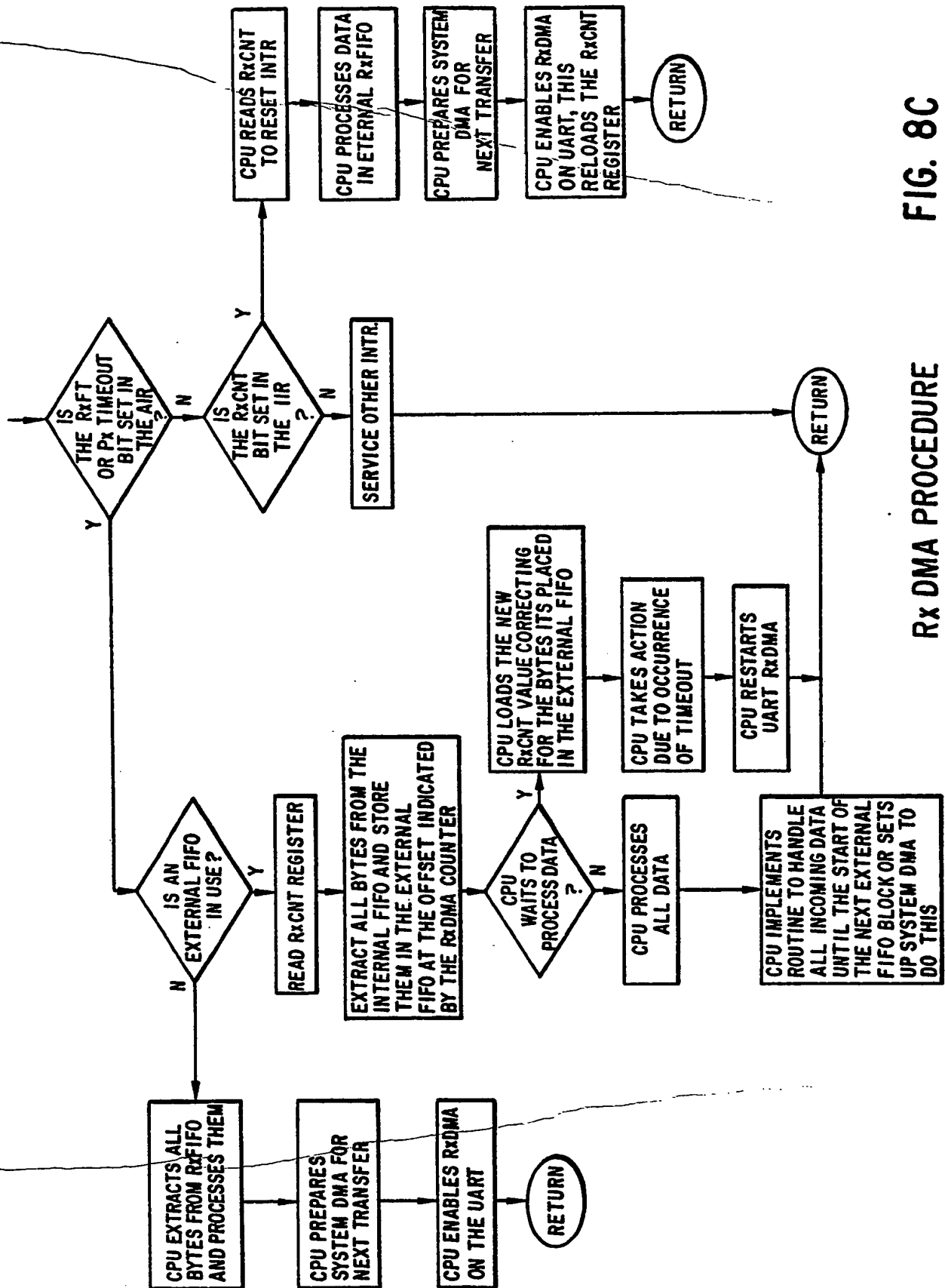


FIG. 8C

Rx DMA PROCEDURE



(19)



Europäisches Patentamt
European Patent Office
Office européen des brevets



(11) Publication number:

0 359 137 A3

(12)

EUROPEAN PATENT APPLICATION(21) Application number: **89116585.4**(51) Int. Cl.⁵: **G06F 13/28, G06F 13/38**(22) Date of filing: **08.09.89**(30) Priority: **14.09.88 US 244920**(43) Date of publication of application:
21.03.90 Bulletin 90/12(64) Designated Contracting States:
DE FR GB IT NL(88) Date of deferred publication of the search report:
06.11.91 Bulletin 91/45

(71) Applicant: **NATIONAL SEMICONDUCTOR CORPORATION**
2900 Semiconductor Drive P.O. Box 58090
Santa Clara California 95051-8090(US)

(72) Inventor: **Michael, Martin S.**
2301 Flint Avenue
San Jose, CA. 95148(US)

(74) Representative: **Sparing - Röhl - Henseler**
Patentanwälte
Rethelstrasse 123 Postfach 14 02 68
W-4000 Düsseldorf 1(DE)

(54) **Universal asynchronous receiver/transmitter.**

(57) Data characters to be transferred from a peripheral device to a central processing unit are serially shifted into the receiver shift register of a universal asynchronous receiver/transmitter (UART). A multiple byte first-in-first-out memory stores a plurality of data characters received by the shift register. The UART checks the status of each data character stored in the FIFO to determine whether it will trigger an exception. A bytes till exception register indicates the number of data characters remaining in the FIFO until an exception is encountered. Then, upon request by the CPU, the UART provides the count of consecutive valid data characters from the top of the FIFO to the first exception, eliminating the need to check status on every transferred byte. Each of the multiple channels of the UART includes an Initialization Register. Setting the appropriate bit Initialization Register of any UART channel allows concurrent writes to the same selected register in each channel's register set. This function reduces initialization time for all of the common parameters that are loaded into each channel's registers. The UART implements a methodology which allows for the processing of any control characters or errors received by the UART during DMA while internal and/or external FIFOs are being used.

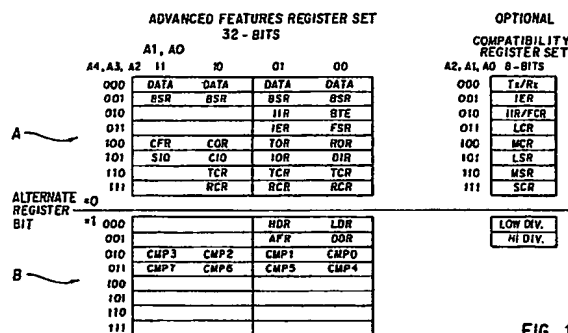


FIG. 1

REGISTER SET OVERVIEW (8, 16, 32 BITS)

EP 0 359 137 A3



European
Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 89 11 6585

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl.5)
A	EP-A-0 029 800 (UNITED TECHNOLOGIES CORPORATION) * page 2, line 34 - page 3, line 31 ** claim 1 ** figure 1 *	1-32	G 06 F 13/28 G 06 F 13/38
A	US-A-4 368 512 (S. KYU ET AL.) * column 12, line 41 - line 68 ** column 15, line 10 - line 35 @ figures 7-8 *	1-32	
A	EP-A-0 125 561 (IBM CORPORATION) * page 1 ** page 3 - page 5 *	1-32	
A	EP-A-0 089 440 (IBM CORPORATION)		
A	US-A-4 346 440 (MOTOROLA INC)		
The present search report has been drawn up for all claims			TECHNICAL FIELDS SEARCHED (Int. Cl.5) G 06 F
Place of search The Hague		Date of completion of search 09 September 91	Examiner NGUYEN XUAN HIEP C.
<div>CATEGORY OF CITED DOCUMENTS</div> <div>X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document T : theory or principle underlying the invention</div> <div>E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document</div>			

Patent Abstracts of Japan

PUBLICATION NUMBER : 60211559
PUBLICATION DATE : 23-10-85

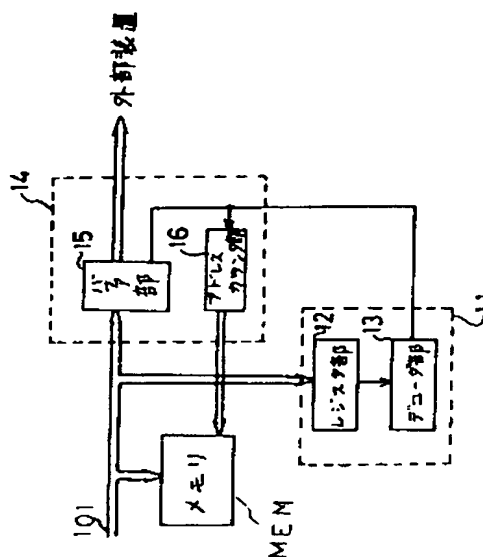
APPLICATION DATE : 06-04-84
APPLICATION NUMBER : 59068667

APPLICANT : NEC CORP;

INVENTOR : SUGITA AKIHIRO;

INT.CL. : G06F 13/38

TITLE : COMMON MEMORY CONTROL SYSTEM

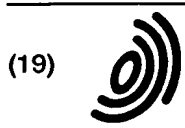


ABSTRACT : PURPOSE: To use more efficiently a memory by adding a specific code before and after a data to be changed to prevent competition when the memory is used in common asynchronously and a computer revises the content of the memory.

CONSTITUTION: When the computer rewrites the data of the memory MEM, the specific code (SKIP and END) is added before and after the data to be modified and the result is transmitted to a data bus so as to revise the content of the memory. When the specific code SKIP is detected by a decoder section 13, it is transferred to a transfer control section 14, which stops the transfer of data. When the revision of the memory is finished, the decoder section 13 detects the specific code END, gives it to the control section 14, which restarts the data transfer. In this case, an address signal outputted by an address counter section 16 is not affected by the stop of data transfer. Further, the computer can revise optionally the content of memory as required. Thus, the competition of the use of the memory is prevented and the memory is used more efficiently.

COPYRIGHT: (C)1985,JPO&Japio

This Page Blank (uspto)



Europäisches Patentamt
European Patent Office
Office européen des brevets



(11) EP 0 784 268 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
16.07.1997 Bulletin 1997/29

(51) Int. Cl.⁶: G06F 9/46

(21) Application number: 96120614.1

(22) Date of filing: 20.12.1996

(84) Designated Contracting States:
DE FR GB NL SE

(30) Priority: 10.01.1996 US 585364

(71) Applicant: SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventors:
• Vasudevan, Rangaswamy
Los Altos Hills, California 94022 (US)
• Jalali, Caveh
Mountain View, California 94041 (US)

(74) Representative: Liesegang, Eva
Forrester & Boehmert,
Franz-Josef-Strasse 38
80801 München (DE)

(54) Generic remote procedure call system

(57) A system and method allow client applications to invoke remote procedures on a server application using any of a plurality of remote procedure mechanisms, by selecting a remote procedure call mechanism at runtime. The system and method uses client and server stubs in the application that include an mechanism-independent canonical specification of each procedure interface. The specification defines the form of the interface and arguments, but not does include conventional mechanism-specific marshalling arguments for marshalling the arguments. The resulting compiled stubs may be used with any remote procedure call engine. Such remote procedure call engines receive the specification of the procedure interface and the arguments passed by the client application to the server, and determine at runtime the particular marshalling routines to use, according to the canonical specification. This defers selection of the marshalling routines, and hence allows a single compiled client application binary code to be used with any of a variety of remote procedure call engines and marshalling routines. Deferring selection of marshalling routines further allows optimization of data types where the client and server computers share architectural characteristics. The system includes a interface definition language compiler that produces the client and server stubs having the canonical specification of the procedure interfaces, a virtual remote procedure library that selects a remote procedure call engine for a client, and plurality of remote procedure call engines.

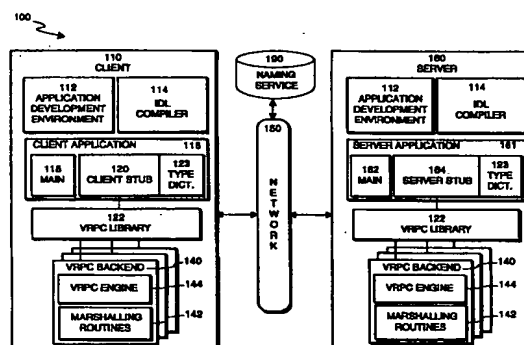


FIGURE 1

EP 0 784 268 A2

Description

BACKGROUND

5 Field of Invention

The present inventions relates to client-server distributed computing systems generally, and more particularly, to systems providing multiple remote procedure call mechanisms dynamically selected at runtime to provide client-server interprocess communication and data transfer.

10

Background of Invention

Client-server computing has become the predominant model of distributed computing, paralleling the increasing performance of desktop computers and workstations. In a client-server distributed computing environment, multiple computers are connected in a network, and a computer may operate both as client, a consumer of resources and data, and a server, a producer of resources and data for the clients. Accompanying the spread of client-server systems, there has been a move away from the homogenous networks common in the past, where there was typically a single or very few vendors providing all the computers, to heterogeneous networks with computers, software, and peripheral devices from multiple vendors. In this environment, integration of client-server operation becomes a critical requirement.

15

In any client-server environment, a client requests operations of a server through a *remote procedure call* (RPC). In a remote procedure call, a process on a local computer, the client, invokes a process, the server, on a remote computer. Historically, the idea was to perform the remote procedure call with a single thread of control held by the client, so that the RPC behaved in the same manner as a local procedure call. In order to achieve this goal, there must be an agreed upon set of semantics between the client and the server for describing the procedure call and specifically, the arguments passed across the call. This is because both the client and the server may have different internal architectures for representing data, and thus, explicit specification of types of arguments is used to communicate data between the client and server. In addition, because the client and server typically are different computers, each with its own address space, passing parameters by reference is not usually possible. Conversion and construction of data for transmission between the client and server is called *marshalling*. In a general purpose RPC mechanism explicit typing is necessary, since implicit typing cannot be used unless a single RPC mechanism is chosen. In a heterogeneous environment, there may be multiple RPC mechanisms in use, and thus, explicit typing is needed. However, marshalling routines are specific to each RPC mechanism, since they designed to construct the data for a particular and machine architecture RPC mechanism.

20

25

30

In conventional RPC systems, the client holds an interface to the server in the form of a client stub. The server has a server stub that provides the linkage to the server application. These stubs are created by the application developer by using the interface definition language (IDL) to specify the interface, and then compiled by an IDL compiler. The IDL is used to specify the interface between the client and the server. The client-server interface is conventionally defined as a set of procedures with specifically defined input and output arguments.

35

In conventional systems, the semantics of each remote procedure call are implemented when the stubs are compiled by the IDL compiler. The IDL compiler determines the specific marshalling routines for use the arguments in a given procedure, and links these routines into the object code of the interface. The stub is then compiled into the application binary. The resulting, executable binary application code thus, can only be used with the particular RPC mechanism that uses the linked-in marshalling routines. At runtime, there is no ability to select which RPC mechanism to use when several are available, since the marshalling routines are already part of the application.

40

In a homogenous computing environment where there is a single RPC mechanism, this approach was acceptable. The selection of a single RPC mechanism ensured that all applications were developed to use just that mechanism. However, homogenous systems are no longer the typical environment. Rather now heterogeneous systems are commonplace where a single RPC mechanism becomes a limitation on the interoperability of clients and servers. Ideally then, a client application should be able to use the services of any server, regardless of the RPC mechanism. That is, where multiple RPC mechanisms are available on the system, the client should be able to select an RPC mechanism when the server is invoked.

45

50

With conventional RPC mechanisms, this is not possible because conventional RPC systems are incompatible and have incompatible APIs. While the individual RPC, mechanisms in and of themselves support heterogeneous environments for different machine architectures, clients and servers do not support heterogeneous operating environments for different RPC mechanisms. More specifically, there are numerous distinct interface definitions languages, each with its own semantics, and with distinct compilers. Each IDL compiler is exclusively tied to distinct RPC mechanism, and each RPC mechanism has its own particular set of marshalling routines for servers using the RPC mechanism. Marshalling routines of one RPC system cannot be used with another. Thus, existing applications that are compiled for use with a specific RPC mechanism cannot be used with other RPC mechanism. This prevent an application selecting an

55

RPC mechanism at runtime.

To make the application interoperable with other RPC mechanism conventionally requires rewriting the interface of the server in the specific IDL of the new RPC, and then modifying and recompiling the application with the marshalling routine of the RPC. This process is expensive and time consuming, and results in the user having to choose which application to execute depending on which RPC mechanism is desired. The selection of RPC mechanisms is confusing to the user and inefficient.

Accordingly, it is desirable to use a mechanism that isolates the interface definition of the client-server interface from the RPC mechanism and marshalling routines, so that the RPC mechanism can be dynamically selected when the client is about to use the RPC mechanism on the server. Since the selection is performed at runtime, the client and server can select the RPC mechanism that best suits the operating environment on a per invocation basis. This allows the selection of RPC mechanism to be made at runtime, rather than at compile time, resulting in the desired heterogeneous system flexibility. Deferring selection of marshalling routines to invocation also makes clients and server more portable between hardware platforms and operating systems and allows the addition of new RPC mechanism to an installed base without change to such mechanism.

SUMMARY OF THE INVENTION

In one of its aspects, the present invention overcomes the limitations of conventional RPC systems by creating a canonical specification of a procedure interface at the time that the client or server stub is compiled. When the client stub is subsequently invoked to initiate the remote procedure call, this canonical specification is passed to a selected RPC engine which in turn determines how to marshal the arguments used in the interface and invoke the call. Only when the RPC engine is selected are the marshalling and invocation routines determined. Interpreting the canonical specification at runtime allows for the most optimal implementation of the specification, rather than fixing any form of the implementation at compile time. The resulting implementations from the invention, because they derive directly from the canonical specification, they will offer higher performance than implementations derived from predetermined set of marshalling and invocation routines.

With the interpreted canonical specification, RPC engine marshalls the interface arguments (if and when necessary) and transmits them to the server by invoking the server stub. The server stub then unmarshalls data, interprets the canonical interface specification to determine the actual arguments and data types, executes the call, and returns the result arguments.

This process provides a "virtual" remote procedure call ("VRPC") system. The remote procedure call is "virtual" because the client stub does not have the specific implementation that marshalls the argument at the time of the call. The virtual remote procedure call system separates out the semantics of the procedure call, which are important to the RPC mechanism, from the syntax, which is important to the application developer. In addition, because the determination of RPC engine and the selection of marshalling routines is delayed until runtime, the canonical specification may be optimized where the client and server computers are of the same architecture, by using opaque data types rather than marshalling into machine independent representations. This optimization further improves the performance of the RPC mechanism.

In one aspect of the invention, the canonical specification is created by an interface definitional language compiler when the stubs are generated. The canonical specification describes the number of arguments, the data type and at least one argument mode in a machine independent manner. In one embodiment, the specification of data type is provided using an agreed set of bytecodes that define the representation format of the canonical specification. Complex data types are specified recursively. The actual data values for the interface arguments are not included in the canonical specification, but determined at runtime. The use of bytecodes enables the VRPC system to be independent of specific RPC mechanisms since the bytecodes define the argument datatypes independently of the machine architecture on which an arbitrary RPC mechanism is implemented. For example, a bytecode specifying an integer datatype may define it to be precisely two bytes in length. In contrast, in conventional programming languages, such as C, the programmer can define a variable to be an integer, but cannot specify that all integers are two bytes. Rather, the computer on which the program is compiled has its own internal format for storing the data and stores the integer in that predetermined format, and that storage format may be different from other integer formats on other computer architectures, even for the exact same program. With the bytecode specification of the present invention, regardless of how the client or server computer internally define an integer, it is stored in the same memory format.

Another aspect of the present invention is a virtual remote procedure call library and a number of virtual remote procedure call mechanisms, or "VRPC backends." Each VRPC backend corresponds to a specific RPC system and includes a virtual remote procedure call engine that performs that actual transfer of data across the network, and a set of marshalling routines. The VRPC library facilitates communication between a client stub and a server, and enables the selection of VRPC backend by a client stub.

Because the selection of RPC mechanisms is deferred until runtime, the use of the VRPC system on a given client machine can be used to select any available RPC mechanism without the need for the VRPC system to be present on

other server machines on the network. This provides various advantages, such as allowing any arbitrary RPC mechanism to work with the VRPC system, and thus allowing the client program to use any RPC mechanism without upgrading the server or recompiling the client program. In addition, the VRPC system allows individuals client machines to be upgraded over time, without requiring all clients on a network to use the VRPC system. This reduces the cost and difficulty of migrating a large network to the VRPC system.

In another aspect of the invention, the use of canonical specifications enables the generation of wire-compatible protocols with existing and unmodified RPC servers of various types. Furthermore, existing interface definition language compilers can be modified to produce VRPC stubs without requiring any change in the underlying programming language. In addition, existing network and RPC-specific libraries may be used to guarantee RPC protocol compatibility.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is an illustration of a client-server distributed computing system in accordance with the present invention
Figure 2 is a data flow diagram of the process of generating client and server stubs.

Figure 3a is an illustration of a type hierarchy for the interface specification.

Figure 3b is an illustration of the send call invocation of a VRPC engine.

Figures 4a and 4b are an event trace and data flow diagram of the process of invoking a VRPC engine.

DETAILED DESCRIPTION OF THE INVENTION

System Architecture

Referring now to Figure 1, there is shown one embodiment of the virtual remote procedure call system of the present invention in a client-server distributed computing system. The VRPC system 100 is a distributed client-server system capable of operating with heterogeneous machine architectures. System 100 includes client computers 110 (one client computer 110 is illustrated), coupled to server computers 160 through a network 150. The client computers 110 may be conventional personal computers or workstations, such as Sun Microsystems Inc. SPARCstations, Intel-based x86 computers, IBM 390 or 400 series microcomputers, or the like. In the preferred embodiment, the client and server computers use Sun Microsystems' Solaris 2.x operating system, including the Open Network Computing Plus (ONC+) networking environment.

The client computers 110 provide both an application development and execution environment for developing application programs in accordance with the present invention. The application development environment 112 includes application programming tools such a code editor, a source code compiler, a linker, a syntax checker, and the like. In accordance with the present invention, an IDL compiler 114 is also provided. The IDL compiler 114 generates client and server stubs that are independent of any rpc mechanism. More particularly, a client stub has no dependencies on interface or network services that defines the marshalling of the input and output arguments of the client stub.

Rather, the client stub 120 and server stub 164 both include a canonical specification of their interfaces. The canonical specification defines in a machine and RPC-independent manner the data structures and interface routines for the client and server interfaces. The canonical specification uses explicit typing of data structures and interface routines. In the preferred embodiment, the explicit typing is provided by a bytecode specification that is understood by a variety of VRPC backends 140 and marshalling routines 142. The canonical specification is machine readable and can be executed by any VRPC engine 144 to communicate a data stream containing the arguments and data types of the interface between the client and server for a given remote procedure call. A type dictionary 123 created by the IDL compiler 114 and provided in the client 116 and server applications 161 stores the specific types used in the client and server stubs.

The bytecode specification in the client and server stubs are conventionally compiled with main application source code routines to generate client applications 116 and server applications 161. As compiled, stubs are executable by the application in a normal manner since the stubs have the appropriate syntax for interfacing with their applications.

In the execution environment, there is provided in the client 110 and server 160 computers a VRPC library 122, and a plurality of VRPC backends 140. The VRPC library 122 provides the communication linkage between the individual client stubs 120 and the server stubs 164 and one of the VRPC backends 140. The VRPC library 122 includes various routines that are invoked by either the client stub or server stub, as necessary to select a VRPC backend 140 and establish a connection therebetween. The routines dynamically select an available VRPC backend 140, passing to it the canonical representation of the interface and the data used by the stub. The selection process is determined at runtime, rather than during compilation of the stubs, as is done conventionally. The VRPC library 122 includes a service provider interface for the VRPC backends 140, so that any VRPC backend 140 can be substituted for any other.

Each VRPC backend 140 includes a VRPC engine 144 and marshalling routines 146. The VRPC engine 144 receives the canonical specification of the interface provided in the client 120 or server stub 164, along with handles to the data being passed, and calls the appropriate marshalling routines 142 to format the data structures and interface routines according to the bytecode specification in the canonical specification, passing the data values to the marshal-

ling routine 142 since the marshalling routines 142 on the client share the same address space. The bytecode specification only defines the data structures of the interface, it does not specify how these are to be marshalled. Rather, each VRPC engine 144 is responsible for determining which marshalling routines 142 are used. In a preferred embodiment, there are multiple different VRPC backends 140, each specific to a single rpc mechanism, such as ONC-RPC supported by Sun Microsystems, or DCE-RPC, or the like.

The marshalling routines 142 marshal and unmarshal the passed in arguments, and return them to the VRPC engine 144. These routines are dynamically determined and invoked at runtime, rather than linked into the application binary itself. In this manner, the VRPC engine 144 is selected only when the client is invoked, not when the client stub 120 is generated by the IDL compiler 114 as in conventional systems.

A naming service 190 provides name to address translation for server applications 161 on the network 150. Each server application 161 on the network 150 has an address, maintained in the naming service 190, and optionally including a specification of which VRPC backend 140 is to be used with the server 161. The server applications 161 are preferably responsible for registering themselves with the naming service 190. The naming service 190 provides the name or address of a server in response to requests from other entities, such as the VRPC library 122 or client application 116.

Because the selection of VRPC engine 144 and marshalling routines 142 is deferred until runtime, any number of different VRPC engines 144 and marshalling routines 142 can be used with the same client 116 and server stubs 161, without having to recompile the stubs and application for the new rpc mechanism. This allows a single application binary to be distributed and be known to be compatible with current or future rpc mechanisms available on the network, thereby eliminating the cost and difficulty of converting each application to each specific rpc mechanism.

System Operation

Generation of Client and Server Stubs

Referring now to Figure 2, there is shown a data flow diagram of the process of generating client 120 and server stubs 164. For the purposes of this disclosure, the present invention is described with respect to a development and execution of a single client stub 120 for a single application. Application to the general case will be understood by those of skill in the art.

The application developer, using the application development environment 112 produces a client application 116 and server application 161 generally as follows. The developer writes a client code 206 and server procedure code 202 in a conventional manner.

The developer further writes an interface specification for the client and server applications in an IDL language compatible with the IDL compiler. The interface specification defines a set of functions or procedures of the client and server, along with their input and output arguments. In accordance with the invention the, IDL compiler 114 is independent of the RPC mechanism, and so any IDL language compiler is acceptable. Suitable IDL languages includes DCE/RPC IDL, Microsoft's MIDL, CORBA IDL, Sun Microsystems ONC-RPC IDL. In Figure 2, the interface specification file is illustrated generally by the file foo.x (204).

The IDL compiler 114 compiles the IDL specification file to produce a client stub 120, here foo_client.c and separately a server stub 164, foo_server.c. The stubs are source code files in a target source language and provide the source code level interfaces for the client and server.

More particularly, the IDL compiler 114 provides three functions. First, it parses a client-server interface specification described in the input IDL file. The typical IDL language is similar to procedural programming languages, but general provides only structure declarations, function prototypes, and constant expressions. In the preferred embodiment, the IDL compiler 114 may include a number of different front end parsers, each one adapted for handling a different interface definition language.

Second, the IDL compiler 114 generates programming language specific bindings for the interface specification. This involve mapping parsed interface definitions to the specific programming language and operating system constraints of the execution environment. The language binding of an IDL describes how a particular function or data structure is translated from the IDL language to the target programming language. The output of the IDL compiler 114 is a source file providing a specification of the interface in a form that can be compiled with, and executed by the client and server applications. The preferred target source languages are C and C++.

Finally, the IDL compiler 114 generates the actual client and server stub code for the language bindings that performs the actual remote procedure calls. Unlike conventional IDL compilers 114 that produce stub code that includes calls to specific marshalling routines for marshalling the data structures of the interface, the stubs produced by the IDL compiler 114 do not specify the marshalling routines to be used for passing the input and output arguments. Rather, the stubs include the canonical specification of the data structures being passed between the client and the server. The canonical representation is maintained in a first data structure, an interface definition structure. The interface definition structure provides an abstract description of the format of the interface, including the data types of the number of input

and output arguments, their number, and an identification of the remote procedure. The specification of the data types is provided by a bytecode; each of the VRPC engines 144 is capable of interpreting the bytecodes and determining the proper marshalling routines. A separate data structure or set of data structures is used to point to the argument values themselves.

5 The IDL compiler 114 further includes in the client stubs 120 calls to the VRPC library 122 and an abstract VRPC engine 144. These calls pass the interface specification to the VRPC library 122 and VRPC engine 144. Rather, the VRPC library 122 calls a selected VRPC engine 144 in one of the VRPC backends 140, to interpret or compile the implementation of the client interface, and the VRPC engine 144 performs the transformation of the specification into its implementation using its associated marshalling routines. The IDL compiler 114 also includes in a header file for the
10 client stub 120 calls to establish and terminate a connection with the server 161.

The IDL compiler 114 further produces in the client stub 120 and the server stub 164 a type dictionary 123 that contains all of the data types used in the interface definition. Each data type in the type dictionary 123 is a defined by a series of bytecode sequences. This allows complex data types in the procedure interface to be represented by references to entries in the type dictionary 123. On the server side, the type dictionary 123 is used by the server stub 164
15 to reconstruct the data types of the raw data received during the remote procedure call.

Figure 3a illustrates the type hierarchy of a preferred embodiment of the interface specification 324. The interface specification 324 is illustrated abstractly as an argument to an invocation of a send method of a VRPC engine 144. This invocation is used by the client stub 120 to invoke a selected VRPC engine 144 to make the actual remote procedure call to the server 161. In this embodiment, the interface specification 304 includes a description of the remote procedure
20 interface and a description of the arguments to be supplied to the remote procedure. The description of the remote procedure interface includes:

an identifier 326 for a procedure number of the particular function procedure. A server interface will typically include a number of procedure calls available from the server. The IDL compiler 114 assigns each of these a unique identifier 326 so that the client and server can reliably exchange information about a specific procedure;
25 a variable 328 specifying the type of the return argument value for the remote procedure. This allows the remote procedure to determine the proper marshalling routine for the return value before sending it to the client; and, a variable 330 specifying the number of arguments between passed in the procedure call.

30 Appendix A includes an exemplary implementation of the type hierarchy of Figure 3a. In that embodiment, the interface specification 324 is provided in the VRPC_proc_spec_t structure definition.

In the preferred embodiment the canonical description of the arguments is an argument specification 332 that includes a set of elements that describe each argument. The actual values of the arguments are not included in the argument specification 332, but only their form. For each argument, there is an argument type 336, and at least one
35 argument mode 334. An argument mode 334 specifies the semantics for handling the arguments between the client and server. Generally, the modes define whether an argument is an input or output argument. In the preferred embodiment, the modes are refined to handle different types of relationships and optimization between the client and server. In a "lend" mode, the argument is provided to the server 161, which modifies it and returns the modified value back to the client 120. In a "lend to" mode, the server 161 receives a copy of the argument, and may use that value during the
40 invocation of the remote procedure, and subsequently deallocates memory for that use. The client must still maintain the argument locally. In a "copy to" mode, a copy of the argument is sent to the server 161, which must free its copy upon completion. In a "copy from" mode, the client 116 does not send data to the server 161, but rather receives data having the defined data type. In a "move to" mode, the argument held by the client 116 is deleted after being sent to the server 161, and the server 161 frees this copy after use. Finally, in a "move from" mode, the server 161 passes data
45 to the client 116 and deleted by the server, and the client 116 frees this instance after use.

An argument type 336 specifies data type for the argument(s). The data type is provided by a bytecode value. The bytecode is used to specify both primitive types, such as ints, chars, and the like, and more complex types that are recursively defined. Appendix A, Section 3.4 includes an exemplary list of bytecodes for representing primitive, pointer, and composed (complex) data types. Other bytecode specifications may also be used. These bytecodes are recog-
50 nized by the VRPC engines 144, and used by the VRPC engines to select marshalling routines to marshal the arguments into the proper representation for transmission to the server, and then reconstruction by the server.

In the preferred embodiment, the argument specification structure 332 is included in the procedure interface specification 324, however, in alternate embodiment, the argument specification structure 332 may be separated from the procedure interface specification. Figure 3a illustrates this relationship. Appendix A provides an exemplary implemen-
55 tation of the argument specification 332, in the VRPC_args_spec_t structure.

Because the selection of marshalling routines according to the data types of the arguments is delayed until the actual invocation of the server interface, the bytecode specification may be optimized on a per-connection basis to take advantage of the architecture of the client and server computers. More particularly, if both the client and server comput-
ers are of the same architecture, such as both Sun Microsystems SPARCstations, then selected data type specific

opcodes can be replaced with opaque data types, thereby eliminating the need to marshal the data into a machine independent format. For example, if the data type specified by an opcode is a double, it may be replaced by eight opaque bytes of data. It is significantly more efficient to transmit the eight opaque bytes than to convert a double into an machine independent representation, such as XDR (see below) and then convert back from the machine independent representation to the double. This optimization may be performed by the VRPC engine 144 that is selected at runtime.

As illustrated in Figure 3a, the interface specification including the argument specification, are passed to an VRPC engine 144 by invoking its send method. When invoked, the VRPC engine 144 traverses the argument specification 332, and determines from the bytecode specification of the argument types 336 the appropriate marshalling routines. The VRPC engine 144 also separately receives the actual argument values 338. The VRPC engine 144 then calls the marshalling routines to marshal the arguments for transmission to the server 161. This process is further described below with respect to Figure 4. The client stub 120 further passes to the VRPC engine 144 a state description 380 (Figure 3) defining the transport parameters for the remote procedure call, and other information, such as the version number 381 of VRPC engine, the name 385 of the VRPC engine, flags 383 for controlling the backend, a handle to the state data 387, and a handle 389 to a destructor function for freeing this state information. An exemplary implementation of the VRPC state description is provided in Appendix A, §4.1.

A simplified example of a client stub 120 created by an IDL compiler 114 illustrating the canonical form of the interface definition is as follows. Assume that the client stub is to contain a function for summing two variables, A and B. In the IDL file *foo.x* this function is represented as:

```
int sum(int a, int b);
```

The IDL compiler 114 creates a canonical specification of this interface in the client stub in the file *client.c*:

```
1. #include "VRPC.h"
2. int sum VRPC_state *be(int a, int b) {
3.   int tmp;
4.   void *arg_addr[3];
5.   arg_addr[0] = &a;
6.   arg_addr[1] = &b;
7.   arg_addr[2] = &tmp
8.   be->send(VRPC_state, inter_descr, arg_addr)
9.   return(tmp);
10. }
```

Line 2 defines the interface to the client stub seen by the client application. The interface is consistent with the application's interface as defined in the IDL file. This allows the application to invoke the client stub without the application developer having to provide any further interfaces between the application and the stub. The body of the stub is hidden from the application, and the application has no information that the sum procedure is being remotely handled. Line 1 specifies the header file which includes the calls for establishing and terminating a connection with a VRPC engine 144. The header file is further described below.

Line 3 defines the output variable, *tmp*. This allows the client to have a local handle to a variable for returning the result from the remote procedure to the application. Line 4 defines an array *arg_addr* that is sized to contain pointers to the two input and one output arguments. In the general case, the argument array contains a pointer for each input and output argument. More generally, in the preferred embodiment, the client stub receives all arguments as pass by reference, including arrays and non-arrays. The argument array corresponds to the argument list 338 in Figure 3a.

Lines 5 through 7 define the contents of the *arg_addr* array, with pointers to the passed in variables *a* and *b*, and a pointer to the output variable *tmp*. In the preferred embodiment, the client stub accepts all arguments as pass by reference, whether these are array or non-array arguments. This arguments array will be passed to one of the VRPC engines and selected marshalling routines. Since these components all reside in the same address space as the client stub, they will be able to access the data values of the argument as need when the data structures of the interface are marshalled at runtime.

Line 8 performs the remote procedure call itself. This call corresponds to the server interface 302 in Figure 3a. The canonical specification of the interface is provided as an input argument *inter_descr* to the *send_call* function, with *inter_descr* being an interface specification 324, *be* is a function pointer to one of the VRPC engines, any of which provide a *send_call* function. The *send_call* function passes down the interface specification 324, *inter_descr*, which here contains a description of the *sum* function. The *send_call* further passes the data that accompanies the interface definition, the *arg_addr* array. When the remote procedure call is completed, the result will be in the temporary variable *tmp*. As shown in Figure 3a, the remote procedure call has a return type 340 that indicates whether the remote proce-

sure call was successful or whether there was an error.

For this example of the function `sum()`, the simplified version of the interface specification 324 would be:

```

5      struct {
          int proc_no = 1;
          int n_args = 3;
10     struct arg_spec args [2] = {(IN, int_bytecode)
                                   (IN, int_bytecode),
15                                   (OUT, int_bytecode)};
          } inter_descr;

```

where *arg_spec* is a structure that implements the argument specification described above. The mode values of "IN" and "OUT" are merely illustrative here, and other argument mode values may be used, as described above. Exemplary mode values are described in Appendix A, §3.2, in the `VRPC_param_mode_t` structure definition. The procedure number, return type, type, and number arguments are determined by the IDL compiler 114.

As noted above, the interface specification 324 does not contain any of the data being passed between the client and server, but merely the form of that data. Because the data are not included in the canonical representation, this representation can be provided to any VRPC engine 144 for marshalling. Only when the marshalling at runtime is actually performed is the data accessed from the arguments array.

The client stub described here is a simplified representation of the interface of the client. Figure 3b illustrates an abstract representation of the interface for this invocation. In a preferred embodiment, the invocation 340 of the VRPC library 122 passes in a data structure, here called `optab_base_t` 353, that encapsulates both the interface specification and argument specification, and an argument list. In addition, the VRPC library 122 receives arguments specifying the version 350 for the client stub 120, flags 352 for setting switches in the library if necessary, the size 354 of the encapsulating structure, for use in marshalling the arguments, a handle 358 to the type dictionary used for the interface, and handles to call the VRPC engine 144 directly for state information (360), its send method 362, and its control method 364. The control method 364 is a general function which the VRPC library 122 uses to instruct the VRPC backend 140 to perform additional functionality. For example, the control method 364 may be used to instruct a VRPC backend 140 to select the security level for a connection between the client and server.

In addition, a set of function pointers 368 to the client stub 120 are also provided for main routine 118 to call. The VRPC engine 144 also receives the argument list 338 with the actual arguments references for the VRPC engine 144 to marshal. The VRPC engine 144 returns a value 340 indicating whether the remote procedure was successful, or whether an error resulted.

When the client application 116 first connects with the server, the client invokes the VRPC library 122 to establish a connection and select a VRPC engine 144. The VRPC library 122 initializes the *be* handle with the address for a selected VRPC engine 144, and client stub function pointers to interface routines. Accordingly, when the client stub 120 is subsequently invoked, it is provided a handle to the VRPC engine 144 which can then marshal the data structures of the interface according to the interface specification structure. The selection process for choosing a VRPC engine 144 at runtime is further described below. The client 116 invokes the VRPC library 122 using interfaces provided in a header file incorporated in all client stubs 120.

The VRPC library 122 provides two basic functions to the clients: initiation/termination of a connection between client/server and selection of a VRPC engine 144. The interfaces to these functions are provided to a client stub 120 in the VRPC header file 209. More particularly, there is a function to establish a connection to the server using a selected VRPC engine 144. This function takes as an input the name of a server to be conducted, generally provided on the server computer 160, and a handle to the client/server interface being accessed. This function invokes the VRPC library 122 on the client computer 110, which obtains from naming service 190 the name and address of a server 161 including a VRPC engine 144 specified for use by the server application 161 having the server interface. The VRPC library 122 then initializes state variable handle *be* for contacting the server using that VRPC engine 144. Otherwise, the VRPC library 122 returns a error value. The VRPC library 122 returns the handle to the VRPC backend 140 to the client stub 120. An exemplary interface for this function is `VRPC_begin()` function described in Appendix A, §1.1. Note that the call to `VRPC_begin()` takes an `optab_base_t` 353 as an argument, thereby providing to the VRPC library 122 the handle to the interface specification and the argument specification in the client sub 116.

A complementary function terminates a remote procedure call connection through the VRPC engine 144 specified the handle obtained from initiation function. This function is called by the application 166 at the end of its execution. An exemplary interface is shown in Appendix A, §1.1 as VRPC_end().

Exemplary implementations of the VRPC library 122 functions are illustrated in Appendix A, §1.1.

Execution of Clients and Selection of a VRPC Mechanism

Referring now to Figures 4a and 4b there is shown an event trace and data flow diagram of the process of executing a remote procedure call in accordance with the present invention. The client application 116 containing the remote procedure call is executed on a client computer 110 in a conventional manner. The application 116 includes a call to initialize the handle in the client stub 120 to the VRPC backend 140 used to invoke the backend's send_call method. The application 116 invokes 400 this initialisation method, for example VRPC_begin(), on the VRPC library 122, passing in an identification of a server interface being accessed. The VRPC library 122 is responsible determining the VRPC engine 144 to be used and establishing a connection to the server application 161 using the VRPC engine 144. The VRPC library 122 queries 402 the naming service 190 for the server address and name of the VRPC backend 140. The naming service 190 returns 404 the address of the server 161 and the name of the VRPC backend 140, which the VRPC library 122 then uses to select the VRPC engine 144. Generally, the selection of the VRPC engine 144 is based on the address of the server or the backend name before the connection to the server is established. The address, or the name retrieved from the naming service 190 identifies the specific RPC mechanism to use, and the library 122 knows how to interpret the address from the naming service 190. The client 116 then connects 406 to the server 161 by calling the VRPC library 122, which then establishes the connection to the server 161 with the selected VRPC engine 144 on the client computer 110, and correspondingly with a VRPC engine 144 on the server computer 160. For example, if the address indicates an ONC-RPC mechanism, the library 122 will load in the ONC-RPC backend, and so on. As a consequence of connection being established, the VRPC engine 144 is loaded by the client computer 110. The VRPC library 122 will obtain a handle to the VRPC engine 144 once the VRPC engine 144 is loaded. The VRPC library 122 returns 407 this handle to the client stub 120 so that future invocations of send_call will be made directly on the selected VRPC engine 144.

At some subsequent point, the main routine 118 invokes 409 the client stub 120 by calling one of the procedures defined therein, and passing in some number of arguments. The client stub 120 packages the actual arguments and the argument specification to create the procedure call, specifying the argument mode(s) and type of each argument, and the procedure description, specifying the procedure number, number of arguments, return type. The client stub 120 then invokes 408 the send_call method. This call is forwarded by the VRPC library 122 to the selected VRPC engine 144 on the client computer 110.

The VRPC engine's 144 on the client computer 110 interprets the interface specification to determine the form of each argument, including its mode(s) and data type according to the bytecode specification. The interpretation further determines from the procedure number the appropriate procedure to invoke on the server 161. From this information, the client's VRPC engine 144 selects the appropriate marshalling routines from the client computer 110 marshalling library 122 for marshalling the arguments into a form that can be transmitted across the network to the server application 161. The VRPC engine 144 invokes 410 these marshalling routines as needed to create the proper representation of an input data structure. Each marshalling routine returns 412 a transport specific data representation for the input data structure. The invocation of the marshalling routines is specific to the VRPC engine 144. In one embodiment based on Sun Microsystems ONC-RPC, the marshalling routines use the data representation specified in XDR: External Data Representation Standard RFC 1104, June 1987. In this environment, the VRPC engine 144 invokes a function clnt_call() of the ONC-RPC mechanism that performs the client side of the remote procedure call for any application:

```
clnt_call(procedure number,
          *datastruct, *XDRfun,
          *datastruct, *XDRfun))
```

This marshalling call specifies the procedure number for the procedure being serviced, and provides pointers to data structures and the marshalling and unmarshalling functions.

Where the client computer 110 and the server computer 160 are of the same architecture, the client VRPC engine 144 may optimize the bytecodes by using opaque data types, rather than marshalling with the marshalling routines.

The VRPC engine 144 on the client computer 110 then sends 414 the marshalled data and arguments to the instance of the VRPC engine 144 on the server computer 160. The VRPC engine 144 on the server computer 160

unmarshalls 416 the arguments using the marshalling library 122 on the server computer 160. Referring to Figure 4B, the marshalling library 122 returns 417 original interface specification to the server's VRPC engine 144.

5 The VRPC engine 144 on the server computer 160 interprets 418 the interface specification, using its type dictionary 123, which corresponds to the type dictionary used by the client side VRPC engine 144. This allows the server's VRPC engine 144 to reconstruct the specific data structures of the original arguments, along with identifying the specific procedure to be invoked in the server application. The VRPC engine 144 passes 419 the arguments in the form specified by the interface specification, and procedure number to the server application 161. The server 161 executes 420 the remote procedure on these arguments to obtain the desired result. The server 161 returns 421 the result to the server VRPC engine 144. The server VRPC engine 144 again marshalls 422, 424 the result according to the interface
10 specification, and sends 426 the marshalled data back to the client VRPC engine 144. The client VRPC engine 144 unmarshalls 428, 430 the data with the client marshalling library 122, and forwards 432 the return arguments to the client stub 120, which passes the result to the client application 116.

15 The VRPC Interface

20
25
30
35
40 Sun Microsystems, Inc. Proprietary information.
Copyright © 1995 Sun Microsystems, Inc.

Table of Contents

5	1 VRPC Client Interface	1
	1.1 Client Initiation	1
	1.2 Client Stub API	2
10	2 VRPC Server Interface	3
	2.1 Server Initiation	3
	3 Interface Specification Mechanism	4
15	3.1 Virtual Data Representation	4
	3.2 VDR Internals	4
	3.3 Type Dictionary	6
	3.4 VDR Bytecode (Type System)	7
	3.4.1 Canonical Types	7
20	3.4.2 Pointer Types	8
	3.4.3 Composed and Meta Types	9
	4 VRPC Backend Interface	10
25	4.1 Interface	10
	4.2 Backend Support Routines	11
	4.3 ONC/RPC Opcodes	12
	4.4 Membuf Opcodes	12
30	Functions Index	14
	Data Types Index	15
35	Concepts Index	16

1 VRPC Client Interface

1.1 Client Initiation

These functions are declared in the following header file:

```
#include <vrpc/vrpc.h>
```

```
void * vrpc_begin (const char *name, const void      Function
                  *clnt_ops)
```

Establish VRPC connection.

name is used to query a naming system to identify and locate the server process.

clnt_ops identifies the interface being accessed. It is the address of a *appl_optab_t* as found in the compiler generated file 'appl_vclnt.c'.

vrpc_begin() returns a pointer to an initialized *appl_optab_t*, or nil on error.

```
int vrpc_end (void *opsp)                                Function
```

Terminate a VRPC connection initiated by *vrpc_begin()*.

opsp is a pointer previously obtained from *vrpc_begin()*.

vrpc_end() returns 0 on success and -1 on error.

```
int vrpc_begin_buf (const char *name, const void      Function
                   *clnt_ops, void *opsp, size_t ops_sz)
```

Establish VRPC connection.

name is used to query a naming system to identify and locate the server process.

clnt_ops identifies the interface being accessed. It is the address of a *appl_optab_t* as found in the compiler generated file 'appl_vclnt.c'.

opsp is the address of an *appl_optab_t* to be initialized. Its size is passed as the next argument. Its size must match the object pointed to by *clnt_ops*.

ops_sz is the size of the structure pointed to by *opsp*.

vrpc_begin_buf() returns 0 on success and -1 on error.

```
int vrpc_end_buf (void *opsp)                            Function
```

Terminate a VRPC connection initiated by *vrpc_begin_buf()*.

opsp is the address of a structure previously initialized by *vrpc_begin_buf()*.

vrpc_end_buf() returns 0 on success and -1 on error.

1.2 Client Stub API

For each remote function defined in the IDL, a client stub routine is generated. Calling this stub function causes the appropriate server function to be invoked with the same arguments; the result of the server routine is returned by the client stub routine.

int opsp->method (appl_optab_t *opsp, args...) Function
Make an RPC call. Invokes function *method* as defined in the IDL.

opsp is a pointer to an *appl_optab_t* as initialized by *vrpc_begin_buf()* (see Chapter 1 [Client API], page 1).

args are the actual RPC arguments. Note that the VRPC client stub routines all accept arguments as pass-by-reference arguments. All non array arguments are passed by reference. Arrays are also passed by reference, but no additional translation is performed since C already passes arrays by reference.

Also note that return function values are passed back in the last argument of the argument list. In other words, the function value is converted to a pass-by-reference variable.

method() returns 0 on success and -1 on error.

Each *'appl_vclnt.c'* file contains an interface definition. The definition is of type *appl_optab_t* and is conventionally placed in a variable called *APPL_optab*.

appl_optab_t Data type

```
typedef struct {
    optab_base_t base;
    int (*func)(appl_optab_t *,
               const func_arg *, int *func_res);
} appl_optab_t;
```

base is of type *optab_base_t* as described below (see Section 3.2 [VDR Internals], page 4).

func each *appl_optab_t* has one or more members which are function pointers. These are initialized by the compiler generated code to client stubs which perform the corresponding RPC.

The *appl_optab_t* has function pointers to the client stubs. Each of these client stubs calls the *be_send_call* routine for that *appl_optab_t* with the appropriate arguments. One of these arguments is a *vrpc_proc_spec_t*.

This page under construction.

2 VRPC Server Interface

2.1 Server Initiation

These functions are declared in the following header file:

```
#include <vrpc/vrpc.h>
```

```
void * vrpc_begin (const char *name, const void      Function
                  *clnt_ops)
```

Establish VRPC connection.

name is used to query a naming system to identify and locate the server process.

clnt_ops identifies the interface being accessed. It is the address of a *appl_optab_t* as found in the compiler generated file '*appl_vclnt.c*'.

vrpc_begin() returns a pointer to an initialized *appl_optab_t*, or nil on error.

This page under construction.

3 Interface Specification Mechanism

3.1 Virtual Data Representation

VRPC uses a Virtual Data Representation scheme to describe application data structures and procedures. Simply put, data structures and procedures defined in an IDL are translated into a specification language which is machine readable. This language consists of specific opcodes (see Section 3.4.1 [bytecode1], page 7) much like a common assembly language.

Since the specification is machine readable, these opcodes can be executed by a VRPC backend to properly communicate a data stream between the client and server for a remote procedure call.

Additionally, specific optimizers can be invoked on a per-connection basis to optimize the byte-code in a way suitable for the architectural characteristics of the client and server machines.

For example, if the peer machines are of the same architecture (eg. SPARC), then the optimizer can replace most data type specific opcodes in the VDR with opaque data types. For example, a double can be replaced with 8 opaque bytes. It is significantly more efficient to communicate 8 opaque bytes than to convert a double into (for example) XDR and then do the translation from the common format back to a double.

3.2 VDR Internals

optab_base_t

Data type

```
typedef struct {
    unsigned long    _version;
    unsigned long    _flags;
    size_t           _size;
    const char       *name;
    const vrpc_tdict_t *vdr_tdict;
    vrpc_be_ops_t    *b_be_handle;
    int              (*b_send_call)(vrpc_be_ops_t *,
                                     const vrpc_proc_spec_t *,
                                     void *ap[]);
    int              (*b_ctrl)(optab_base_t *, int, void *);
} optab_base_t;
```

_version is the version number (1).

_flags option flags (0).

_size is the size of the encapsulating *appl_optab_t*.

Chapter 3: Interface Specification Mechanism

5

name is the string name of the interface described by the encapsulating **appl_optab_t**.

vdr_tdict is a pointer to the VRPC type dictionary for this interface (see Section 3.3 [typedict], page 6).

b_be_handle is a VRPC backend handle.

b_send_call backend routine that makes RPC call.

b_ctrl backend control routine.

vrpc_proc_spec_t

Data type

```
typedef struct {
    unsigned long    opnum;
    unsigned long    rtype_id;
    unsigned long    nargs;
    const vrpc_args_spec_t *args;
} vrpc_proc_spec_t;
```

opnum the procedure number or unique identifier for this procedure within the scope of this interface.

rtype_id the type ID of the return value of this function, or 0 if the return type is void.

nargs the number of arguments this function takes. Also the number of entries in the **args** array.

args an array of **nargs** **vrpc_args_spec_ts**.

vrpc_args_spec_t

Data type

```
typedef struct {
    vrpc_param_mode_t mode;
    unsigned long    type_id;
} vrpc_args_spec_t;
```

mode The parameter passing mode.

type_id The type ID of the argument.

vrpc_param_mode_t

Data type

```
enum {
    PM_LEND,
    PM_LEND_TO,
    PM_COPY_TO,
    PM_COPY_FROM,
    PM_MOVE_TO,
    PM_MOVE_FROM
} vrpc_param_mode_t;
```


PM_LEND The argument is sent to the server, where it is modified, and the modified value is copied back to the client. In a shared memory implementation, this may be implemented as pass-by-reference.

PM_LEND_TO A copy of the argument is sent to the server. The VRPC runtime frees the server's copy of the arguments when the RPC invocation has completed on the server. The client side instance is not freed.

PM_COPY_TO A copy of the argument is sent to the server. The server side application code must free its copy. The client side instance is not affected.

PM_COPY_FROM No data is sent from the client to the server, but data is returned from the server to the client. The server side instance is not freed. The client must free the copy it receives.

PM_MOVE_TO Client side instance is deleted after call. Server side must free instance. In shared memory implementation, the object is not freed, however, responsibility for freeing object moves from client to server.

PM_MOVE_FROM Server side instance is deleted after call. Client side must free instance. In shared memory implementation, the object is not freed, however, responsibility for freeing the object moves from the server to client.

3.3 Type Dictionary

Each 'appl_vdr.c' file contains a dictionary of the types represented in that file. The dictionary is of type `vrpc_tdict_t` and is conventionally called `APPL_typedtab`.

The dictionary is simply an array of pointers to opcode sequences. Each of these opcode sequences defines a data type of the interface. When data types are nested, the reference to the nested data type takes the form of an index into the type dictionary.

vrpc_tdict_t Data type

```
typedef struct {
    unsigned long      nctypes;
    const vrpc_opcode_t *const *ctype;
} vrpc_tdict_t;
```

nctypes Number of type entries in ctype

ctype Each array entry points to a sequence of `vrpc_opcode_ts`. Each of these sequences describes a data type used in this VRPC interface specification.

3.4 VDR Bytecode (Type System)

Each VRPC backend implements a subset of the following opcodes:

3.4.1 Canonical Types

VRPC_bool32 *offset*
 a 32 bit boolean.
 VRPC_byte *offset*
 opaque byte.
 VRPC_int8 *offset*
 8 bit char
 VRPC_int16 *offset*
 16 bit short
 VRPC_int32 *offset*
 32 bit long
 VRPC_int64 *offset*
 64 bit long
 VRPC_uint8 *offset*
 8 bit unsigned char
 VRPC_uint16 *offset*
 16 bit unsigned short
 VRPC_uint32 *offset*
 32 bit unsigned long
 VRPC_uint64 *offset*
 64 bit unsigned long
 VRPC_enum8 *offset*
 an 8 bit enum (eg. C++ support)
 VRPC_enum16 *offset*
 a 16 bit enum
 VRPC_enum32 *offset*
 a 32 bit enum
 VRPC_enum64 *offset*
 a 64 bit enum
 VRPC_float32 *offset*
 a 32 bit float
 VRPC_float64 *offset*
 a 64 bit float

Chapter 3: Interface Specification Mechanism

8

5 VRPC_float128 *offset*
 a 128 bit float
 VRPC_char8 *offset*
 an 8 bit character
 10 VRPC_char16 *offset*
 an 16 bit character
 VRPC_char32 *offset*
 an 32 bit character
 15 VRPC_zchar8 *offset*
 an 8 bit zero terminated character
 VRPC_zchar16 *offset*
 an 16 bit zero terminated character
 20 VRPC_zchar32 *offset*
 an 32 bit zero terminated character

3.4.2 Pointer Types

25 Each of these items describes the type of a pointer. The *offset* is that of the pointer value.

 VRPC_string8 *offset*
 30 a '\0' terminated string of 8 bit chars
 VRPC_string16 *offset*
 a string of 16 bit chars
 VRPC_string32 *offset*
 35 a string of 32 bit chars
 VRPC_string8M *max-len offset*
 a '\0' terminated string of 8 bit chars
 max-len defines length to malloc for string, excluding '\0' terminator
 40 VRPC_string16M *max-len offset*
 a string of 16 bit chars
 VRPC_string32M *max-len offset*
 a string of 32 bit chars
 45 VRPC_pointer1 *ptr-offset type-instr*
 pointer to 1 instance of *type-instr*
 VRPC_pointerN *n-count ptr-offset type-instr*
 pointer to *n-count* instances of *type-instr*
 50

55

Chapter 3: Interface Specification Mechanism

9

5 VRPC_pointerL *len-instr val-offset type-instr*
 pointer to L instances of *type-instr*
 VRPC_pointerML *max-count len-instr val-offset type-instr*
 pointer to *len-instr()* instances in buf of size *max-count*
 10 VRPC_appl
 a pointer to an upcall routine

3.4.3 Composed and Meta Types

15 VRPC_nop
 do nothing.
 VRPC_nest *type-id offset*
 a reference to another type
 20 VRPC_vector *count type-instr*
 fixed sized array of length *count* of type *type-instr*
 VRPC_array
 an array
 25 VRPC_struct *struct-size struct-name*
 a struct
 VRPC_estruct
 end of a struct
 30 VRPC_SunionC *union_size union-offset disc-instr ncases default-type-id*
 (*case, type-id*)
 union-in-struct with cases
 VRPC_RTerror
 35 force runtime error

4 VRPC Backend Interface

4.1 Interface

A VRPC backend is expected to export the following interface. The backend must be implemented as a shared library that the VRPC library can `dlopen()`.

The shared library must have an entry point (function) which uses its arguments to construct and return a pointer to a `vrpc_be_ops_t`.

`vrpc_be_ops_t * vrpc_create_ADDR (const vrpc_tdict_t *ty, const void *addr-info)` Function

`ADDR` is the actual name assigned to the particular VRPC backend.

`TY` is the application's type dictionary (see Section 3.3 [typedict], page 6).

`addr-info` is some backend specific address information. This information may be used to connect to the RPC server.

`vrpc_create_ADDR()` returns the address of an initialized `vrpc_be_ops_t` or nil on error.

Note that the destructor for this data type is called using a function pointer within the data type.

`vrpc_be_ops_t` Data type

```
typedef struct {
    unsigned long  _be_version;
    unsigned long  _be_flags;
    const char     *be_name;
    vrpc_be_state_t *be_data;
    void           (*be_untransport)(vrpc_be_state_t *);
} vrpc_be_ops_t;
```

This type is used to hold state information for a VRPC backend.

`be_flags` is used to hold state flags.

`be_version` is used to indicate the backend version.

`be_name` holds the name of the backend.

`be_data` is used by the backend to store a pointer to its own private state information.

`be_untransport` points to a destructor function for `be_data`. When it is non nil, it is called by the VRPC runtime with `be_data` as an argument when the `vrpc_be_ops_t` is destroyed.

4.2 Backend Support Routines

const op * peek_int (const op *ip, const void *dp, unsigned long *res) Function

This function returns an integer from memory. It is typically used to determine the dimension of dynamically sized arrays and discriminators in unions. The instruction sequence must consist of canonical types.

ip points to the instruction sequence describing the type of integer to read.

dp points to the user's data structure (integer).

res points to where the read integer is returned.

peek_int() returns the value of the advanced *ip*.

const op * poke_int (const op *ip, const void *dp, const unsigned long *val) Function

This function writes an integer to memory. It is typically used to set the dimension of dynamically sized arrays and discriminators in unions. The instruction sequence must consist of canonical types.

ip points to the instruction sequence describing the type of integer to write.

dp points to the user's data structure (integer).

val points to the integer value to be written to memory.

poke_int() returns the value of the advanced *ip*.

size_t size_at (const op *const *tdict, const op *ip) Function

This function returns the size of an object described in VDR.

tdict points to the type dictionary to be used.

ip points to the instruction sequence describing the object.

size_at() returns the size of the object in bytes.

const op * skip1_at (const op *ip) Function

This function advances the instruction pointer to the next VDR instruction.

ip points to the instruction sequence to be skipped.

skip1_at() returns the value of the advanced *ip*.

const op * has_pointer (const op *const *tdict, const op *ip, int *res) Function

This function examines the pointed to object and decides whether the described object contains any pointers.

tdict is the type dictionary.

ip is the address of the VDR description of the object to be examined.
res points to an `int` where this function returns its value. It is set to `TRUE` if a pointer is found, and `FALSE` if no pointers are found.
`has_pointer()` returns the value of the advanced *ip*.

4.3 ONC/RPC Opcodes

The following opcodes are implemented by the ONC/RPC backend module.

```
VRPC_RTerror
VRPC_SunionC
VRPC_bool32
VRPC_byte
VRPC_enum32
VRPC_estruct
VRPC_float{32,64,128}
VRPC_nest
VRPC_nop
VRPC_pointerML
VRPC_pointerN
VRPC_string8
VRPC_string8M
VRPC_struct
VRPC_vector
VRPC_{f,u}int{8,16,32}
```

4.4 Membuf Opcodes

The following opcodes are implemented by the Membuf backend module. Note that the Solaris/Doors backend is built on top of the membuf backend.

```
VRPC_RTerror
VRPC_SunionC
VRPC_bool32
VRPC_byte
VRPC_enum32
VRPC_estruct
VRPC_float{32,64,128}
```

Chapter 4: VRPC Backend Interface

13

VRPC_nest
 VRPC_nop
 VRPC_pointerML
 VRPC_pointerN
 VRPC_string8
 VRPC_string8M
 VRPC_struct
 VRPC_vector
 VRPC_{,u}int{8,16,32}

Functions Index

14

Functions Index

H

has_pointer 11

O

opsp->method 2

P

peek_int 11

poke_int 11

S

size_at 11

skipl_at 11

V

vrpc_begin 1, 3

vrpc_begin_buf 1

vrpc_create_ADDR 10

vrpc_end 1

vrpc_end_buf 1

Data Types Index

15

Data Types Index

A

appl_optab_t 2

O

optab_base_t 4

V

vrpc_args_spec_t 5

vrpc_be_ops_t 10

vrpc_param_mode_t 5

vrpc_proc_spec_t 5

vrpc_tdict_t 6

Concepts Index

B		Opcodes	7
Backend Interface	10		
C		S	
Client API	1	Server API	3
Client Stub API	2		
M		T	
Membuf Opcodes	12	Type Dictionary Layout	6
O		V	
ONC/RPC Opcodes	12	VDR Theory	4

Claims

1. An interface definition language compiler, for use with a computer system providing at least one mechanism-independent remote procedure call system for execution of a procedure in a server, the procedure invoked by a client and taking zero or more arguments, wherein the interface definition language compiler receives an interface definition file including a declaration of the interface of the procedure, and converts it to an mechanism-independent canonical specification including an identification of the procedure in the server, and for each argument included in the interface, a specification of a data type of the argument and at least one argument mode for passing the argument and defining the semantics for using the argument in the procedure, the compiler producing a client stub including the canonical specification, the client stub adapted to be compiled into the client for execution thereby.
2. The interface definition language compiler of claim 1, wherein the interface definition lanaguage compiler produces a server stub adapted to receive the canonical specification of the interface of the procedure, and determine therefrom for each argument a data type and at least one argument mode for the argument for providing the arguments to the server procedure, and for returning an output argument to the client stub.
3. The interface definition language compiler of claim 1, wherein the argument modes include an input mode and an output mode.
4. The interface definition language compiler of claim 1, wherein the argument modes include a lend mode and a lend to mode.
5. The interface definition language compiler of claim 1, wherein the argument modes include a move to mode and a move from mode.
6. The interface definition language compiler of claim 1, wherein the argument modes include a copy to mode and a copy from mode.
7. The interface definition language compiler of claim 1, wherein the canonical specification includes an interface specification comprising:
 - a specification of a procedure identifier that uniquely identifies the procedure in the server;
 - a specification of the number of arguments; and,
 - a specification of a return type for the output argument of the server procedure.

8. The interface definition language compiler of claim 7, wherein the canonical specification includes an argument specification identifying for each of the number of arguments at least one argument mode, and the data type of the argument.

5 9. The interface definition language compiler of claim 8, wherein the data type of each argument is specified by a byte-code representation.

10. The interface definition language compiler of claim 9, wherein:
the interface definition language compiler produces a type dictionary including an entry for each unique data
10 type, the entry including bytecode representation specifying the data type; and,
the argument specification includes for the data type of each argument a reference to an entry in the type dictionary.

11. The interface definition language compiler of claim 1 operating with a computer system comprising:
15 a plurality of marshalling routines;
a client remote procedure call engine and a server remote procedure engine, the remote procedure engines providing a transport mechanism between the client stub and a server stub of the server procedure; and,

20 wherein the interface definition language compiler produces in the client stub an invocation to the client remote procedure call engine to provide thereto the canonical specification, the client remote procedure call engine adapted to invoke selected marshalling routines according to the data types of the arguments specified in the canonical specification to marshal values for the arguments into transport specific structures, the client remote procedure call engine further adapted to provide the transport specific structures to the server remote procedure
25 engine which invokes selected marshalling routines to obtain the argument value and data types, and provides the argument values and data types to the server stub.

12. A computer system for executing in a server application a remote procedure call by a client application, comprising:
30 a client side computer including:

a client application including a client stub to the remote procedure, the client stub receiving zero or more arguments from the client application, and having an invocation that passes the canonical specification to a client remote procedure call engine, the canonical specification of the interface of the remote procedure
35 including:

a procedure identifier uniquely identifying the remote procedure in a server interface of a server;
for any argument specified in the interface of the remote procedure an implementation and machine independent argument specification of a data type of the argument, and at least one argument mode for passing the argument and defining the semantics for using the argument in the procedure; and,
40 a set of references, each reference for obtaining a value to an argument;
a first plurality of marshalling routines, each marshalling routine associated with specific client remote procedure call engine, and adapted to marshal an argument value by constructing for the argument value a representation of argument value that is specific to the client remote procedure call engine and determined by the data type of the argument;
45 at least one client remote procedure call engine, each client remote procedure call engine capable of receiving the canonical specification, and determining from the argument specification of the data type of each argument at the time the remote procedure call is executed, each client remote procedure call engine adapted to invoke for each argument in the canonical specification at least one marshalling routine associated with the client remote procedure call engine to marshal the argument, the client remote procedure call engine invoking the marshalling routine only at a time when the client remote procedure call engine is invoked by client stub, and the client remote procedure call engine determined at substantially a same time as when the client application is executed.

55 13. The computer system of claim 12, further comprising:

a server side computer including:

a server application including the remote procedure;

a server stub providing the interface to the remote procedure;

a second plurality of marshalling routines, each marshalling routine associated with specific server remote procedure call engine to be invoked thereby, and adapted to unmarshall an argument to determine its argument value and data type;

at least one server remote procedure call engine adapted to receive from a client remote procedure call engine each marshalled argument, the server remote procedure call engine invoking at least one marshalling routine of the second plurality to unmarshall the argument to determine the argument data type and argument value, the server remote procedure call engine providing the argument value and data type of each argument to the server stub.

14. The system of claim 13, further comprising:

an interface definition language compiler that produces from the interface of the server procedure the client stub and server stub.

15. The system of claim 14, further comprising:

a type dictionary including an entry for each unique data type in each canonical specification of the server interfaces, a bytecode representation, and the canonical specification of the server interfaces includes for the argument type of each argument a reference to an entry in the type dictionary.

16. A computer implemented method of creating an application stub for compilation into an application, the application stub interfacing the application to a remote procedure call mechanism for executing a procedure call defined in the application stub, the method comprising:

receiving an interface definition of the procedure, the interface definition including zero or more input or output arguments;

converting the interface definition to a canonical specification of the interface by:

specifying an identification of the procedure in the server;

specifying for each argument included in the interface a data type of the argument and at least one argument mode of the argument;

and,

providing the canonical specification in the application stub.

17. The method of claim 16, further comprising:

providing in the application stub an invocation of a client remote procedure call engine, the invocation passing to the client remote procedure call engine the canonical specification.

18. The method of claim 16, further comprising:

creating a server stub to receive the arguments specified in the application stub, and provide the arguments to a server application including the server procedure.

19. The method of claim 16, further comprising:

creating a type dictionary that includes an entry for each unique data type in the canonical specification, the entry including a bytecode representation of the data type, such that the canonical specification includes for the data type of each argument a reference to an entry in the type dictionary.

20. The method of claim 19, wherein the type dictionary is provided to both a client application including the client stub, and a server application including the server procedure.

21. In a computer system including a client computer and a server computer, a computer readable memory for the client computer configured to provide a remote procedure call between the client computer and the server computer, comprising:

a client application including a client stub to the remote procedure,
the client stub receiving zero or more arguments from the client application, and having an invocation that
passes the canonical specification to a client remote procedure call engine, the canonical specification of the
interface of the remote procedure including:

a procedure identifier uniquely identifying the remote procedure in a server interface of a server;
for any argument specified in the interface of the remote procedure an implementation and machine inde-
pendent argument specification of a data type of the argument, and at least one argument mode for pass-
ing the argument and defining the semantics for using the argument in the procedure; and,
a set of pointers, each pointer for obtaining a value to an argument;
a first plurality of marshalling routines, each marshalling routine associated with specific client remote pro-
cedure call engine, and adapted to marshal an argument value by constructing for the argument value a
representation of argument value that is specific to the client remote procedure call engine and determined
by the data type of the argument;
at least one client remote procedure call engine, each client remote procedure call engine capable of
receiving the canonical specification, and determining from the argument specification of the data type of
each argument at the time the remote procedure call is executed, each client remote procedure call engine
adapted to invoke for each argument in the canonical specification at least one marshalling routine asso-
ciated with the client remote procedure call engine to marshal the argument, the client remote procedure
call engine invoking the marshalling routine only at a time when the client remote procedure call engine is
invoked by client stub, and the client remote procedure call engine determined at substantially a same time
as when the client application is executed.

22. The computer readable memory of claim 21, further comprising:

an interface definition language compiler that produces from the interface of the server procedure the client
stub .

23. The computer readable memory of claim 21, further comprising:

a type dictionary including an entry for each unique data type in each canonical specification of the server inter-
faces, a bytecode representation, and the canonical specification of the server interfaces includes for the argu-
ment type of each argument a reference to an entry in the type dictionary.

24. In a computer system including a client computer providing a client application and server computer providing a
server procedure, a computer implemented method of providing a remote procedure call from the client application
to the server procedure, comprising:

invoking the server procedure through an server interface in a client stub in the client application, and providing
to the server procedure a number of arguments;
invoking a remote procedure call engine, and providing thereto a canonical specification of the server interface,
the canonical specification describing for each of the number of arguments, a data type of the argument and
at least one argument mode for passing the argument and defining the semantics for using the argument in the
server procedure;
marshalling each of the arguments according to its data type with marshalling routines selected by the remote
procedure call engine; and,
providing the marshalled arguments to the server procedure.

25. The method of claim 24, further comprising:

selecting the remote procedure call engine at substantially a same time as the invocation of the server proce-
dure from among a plurality of remote procedure call engines according to a designation of a remote procedure
call engine associated with the server procedure.

26. The method of claim 24, further comprising:

determining whether the server computer and the client computer have a common architecture;
selecting arguments from the invocation and providing opaque data types for the arguments instead of mar-
shalling the argument with selected marshalling routines; and,

providing the opaque data types to the server procedure.

5

10

15

20

25

30

35

40

45

50

55

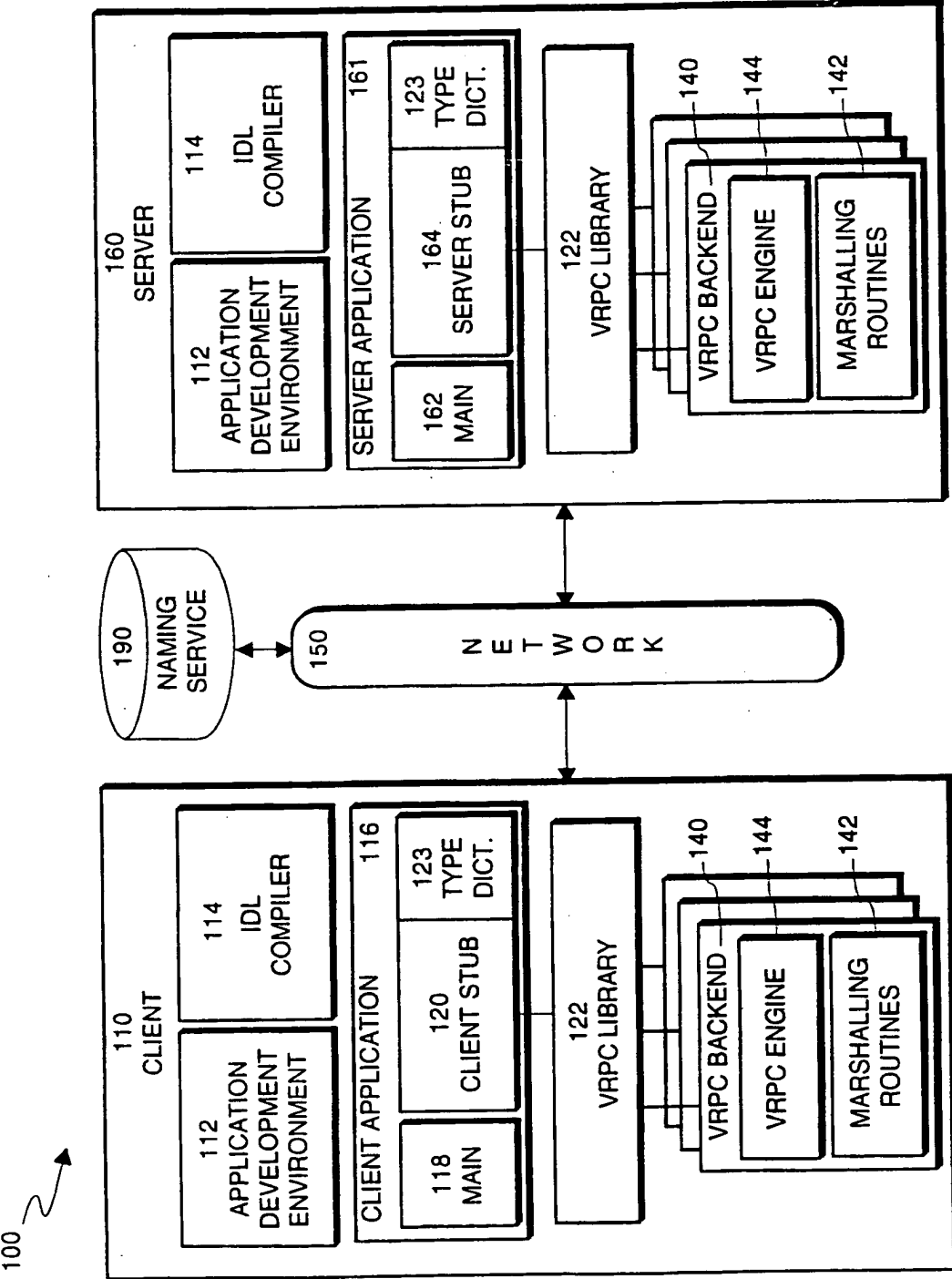


FIGURE 1

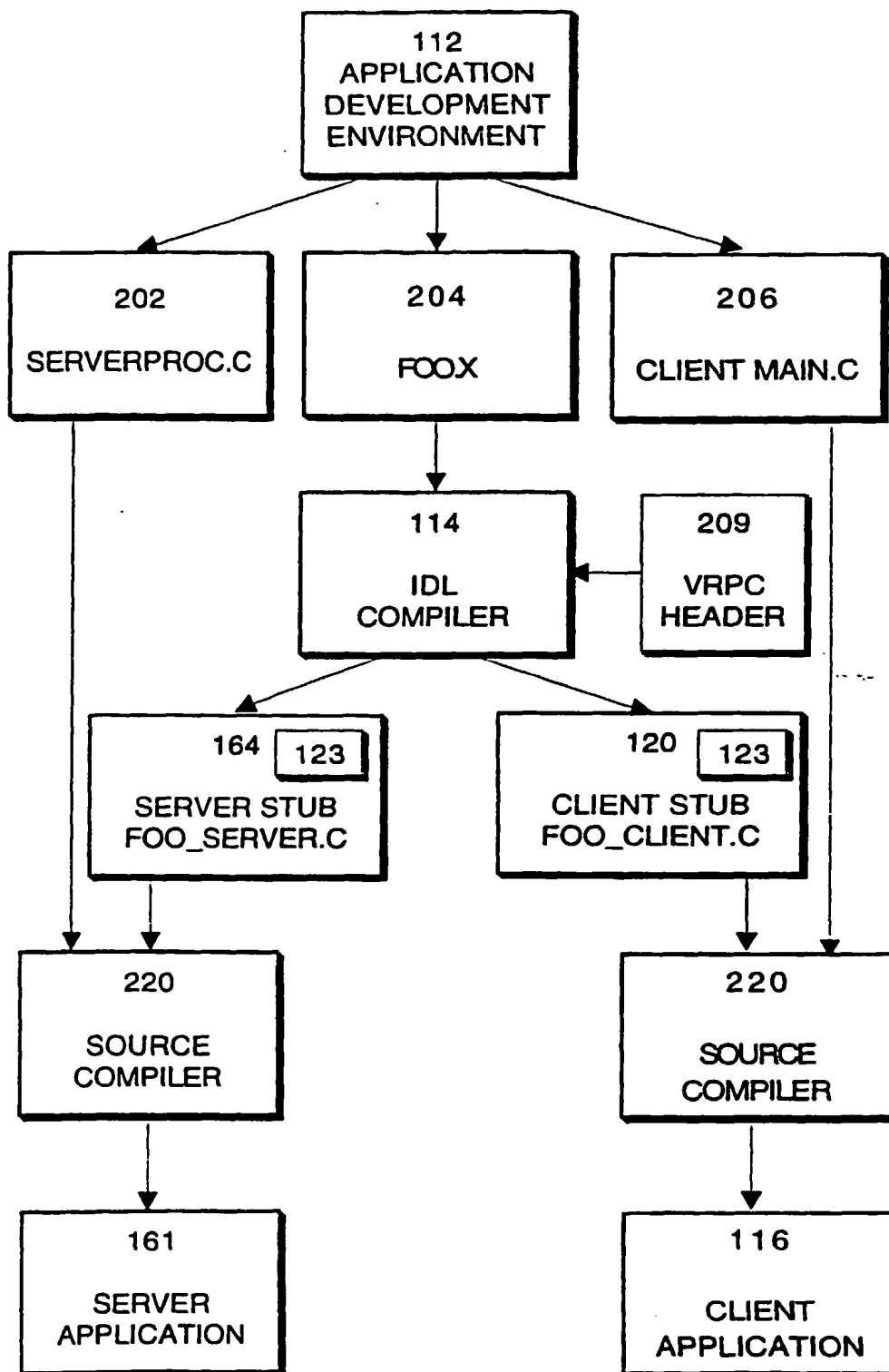
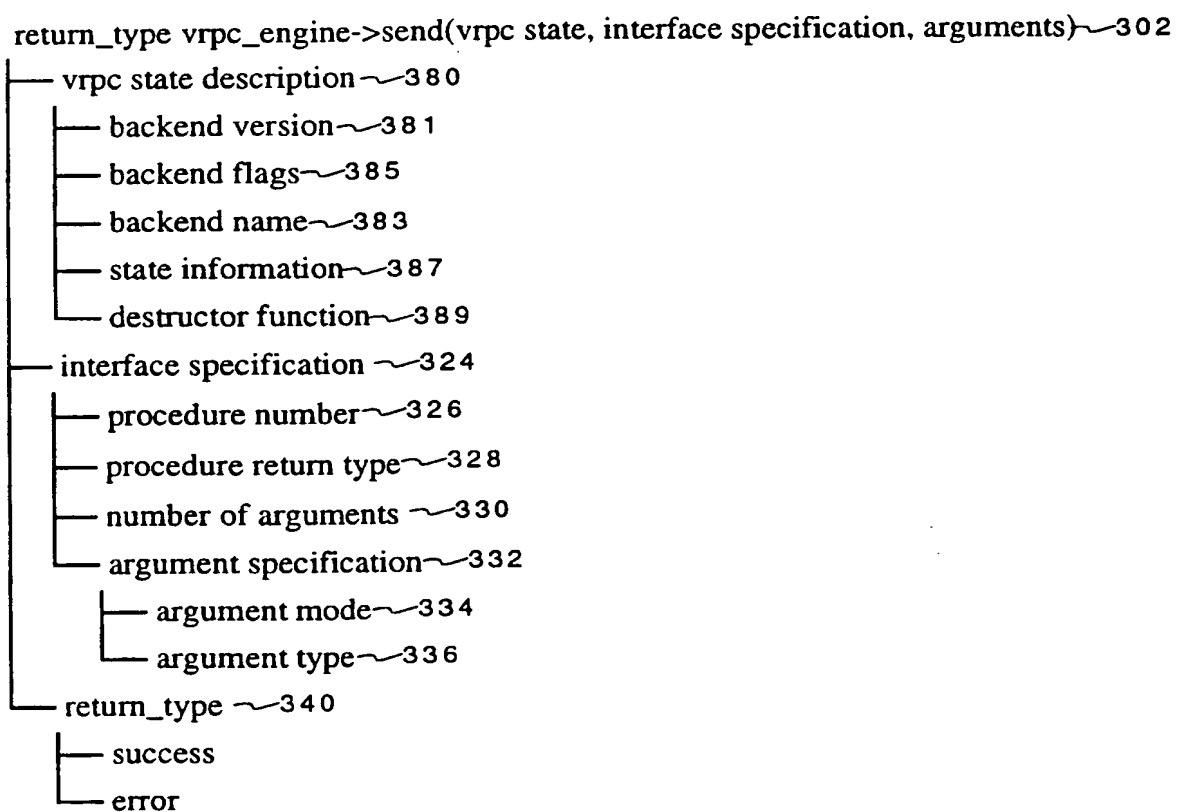
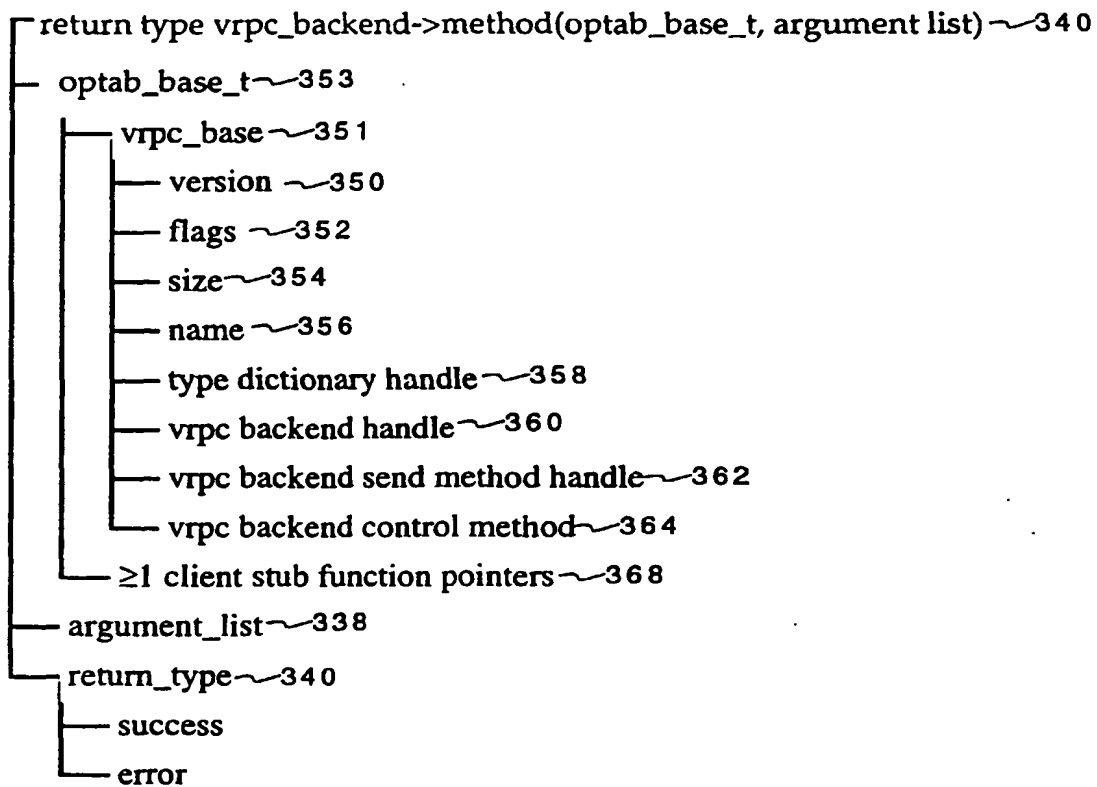


FIGURE 2

*FIGURE 3a*

*FIGURE 3b*

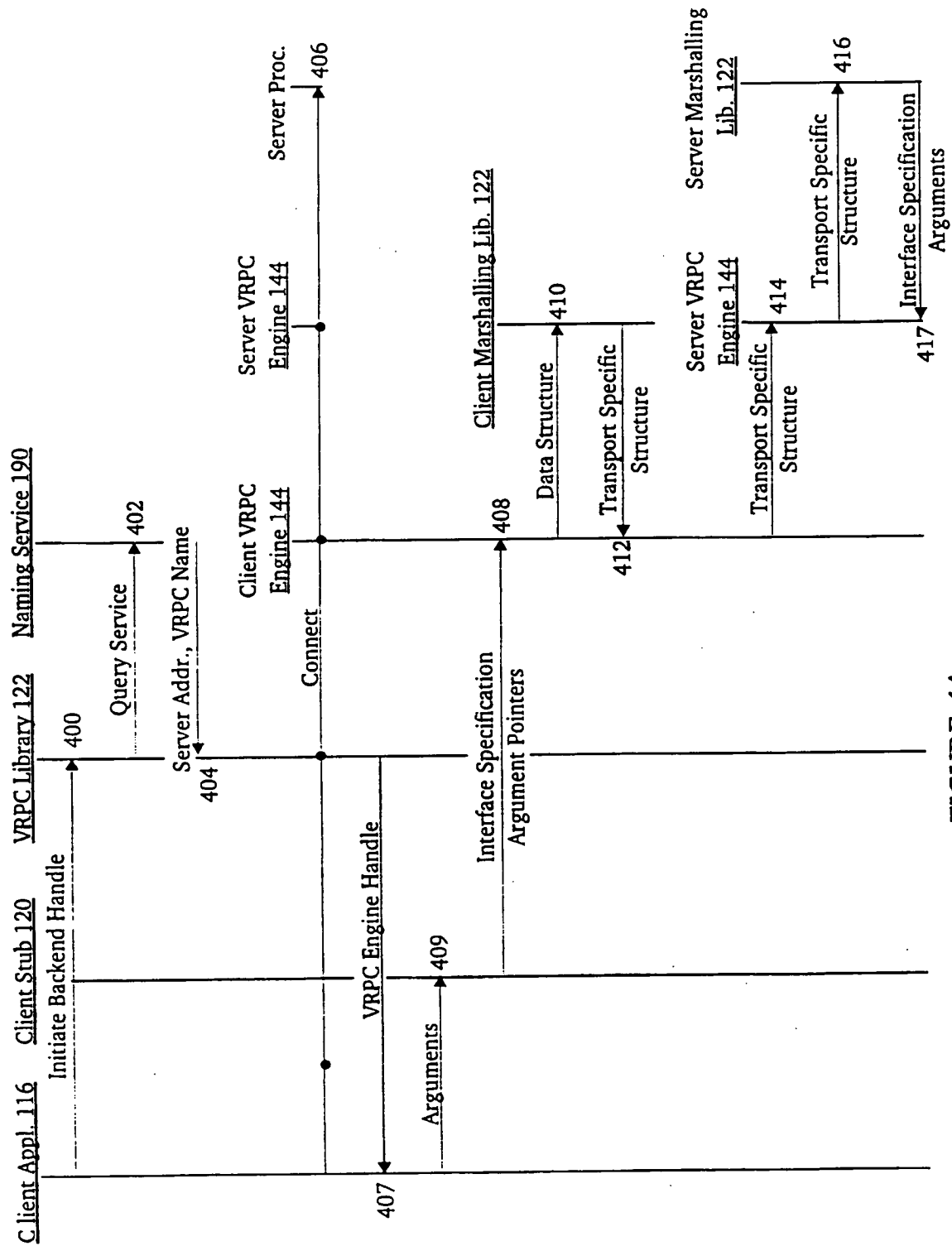


FIGURE 4A

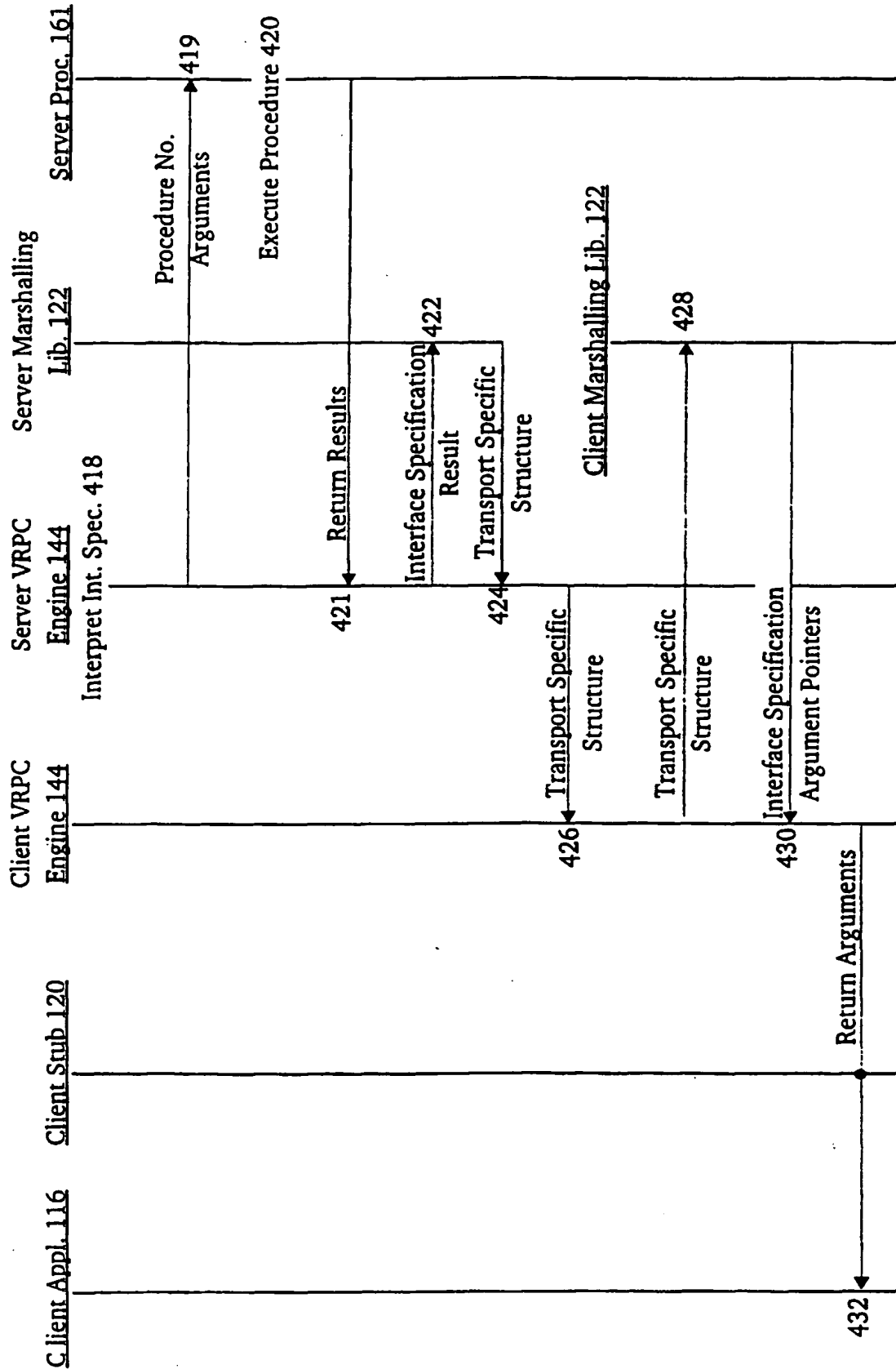
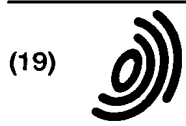


FIGURE 4B





Europäisches Patentamt
European Patent Office
Office européen des brevets



(11) EP 0 784 268 A3

(12) EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
31.05.2000 Bulletin 2000/22

(51) Int. Cl.⁷: G06F 9/46

(43) Date of publication A2:
16.07.1997 Bulletin 1997/29

(21) Application number: 96120614.1

(22) Date of filing: 20.12.1996

(84) Designated Contracting States:
DE FR GB NL SE

(30) Priority: 10.01.1996 US 585364

(71) Applicant:
SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventors:
• Vasudevan, Rangaswamy
Los Altos Hills, California 94022 (US)
• Jalali, Caveh
Redwood City, California 94061 (US)

(74) Representative: Liesegang, Eva
Forrester & Boehmert,
Franz-Josef-Strasse 38
80801 München (DE)

(54) Generic remote procedure call system

(57) A system and method allow client applications to invoke remote procedures on a server application using any of a plurality of remote procedure mechanisms, by selecting a remote procedure call mechanism at runtime. The system and method uses client and server stubs in the application that include an mechanism-independent canonical specification of each procedure interface. The specification defines the form of the interface and arguments, but not does include conventional mechanism-specific marshalling arguments for marshalling the arguments. The resulting compiled stubs may be used with any remote procedure call engine. Such remote procedure call engines receive the specification of the procedure interface and the arguments passed by the client application to the server, and determine at runtime the particular marshalling routines to use, according to the canonical specification. This defers selection of the marshalling routines, and hence allows a single compiled client application binary code to be used with any of a variety of remote procedure call engines and marshalling routines. Deferring selection of marshalling routines further allows optimization of data types where the client and server computers share architectural characteristics. The system includes a interface definition language compiler that produces the client and server stubs having the canonical specification of the procedure interfaces, a virtual remote procedure library that selects a remote procedure call engine for a client, and plurality of remote procedure call engines.

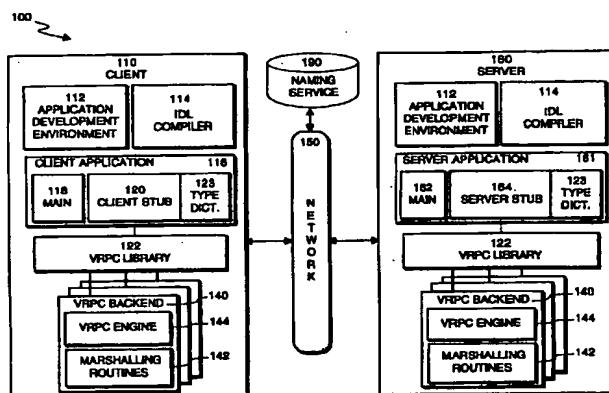


FIGURE 1

EP 0 784 268 A3



European Patent
Office

EUROPEAN SEARCH REPORT

Application Number
EP 96 12 0614

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X	KESSLER P B: "A CLIENT-SIDE STUB INTERPRETER" ACM SIGPLAN NOTICES, US, ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 29, no. 8, 1 August 1994 (1994-08-01), pages 94-100, XP000457339 ISSN: 0362-1340	1, 3, 7-9, 12, 16-18, 21, 22, 24, 25	G06F9/46
Y		4, 6	
A	* page 94, right-hand column, last paragraph * * page 95, last paragraph; figure 1 * * page 96, left-hand column, line 8 - page 97, left-hand column, line 26 * * page 97, right-hand column, paragraph 3.4 * ----- -/--	2, 10, 11, 13-15, 19, 20, 23, 26	
			TECHNICAL FIELDS SEARCHED (Int.Cl.6)
			G06F
The present search report has been drawn up for all claims			
Place of search THE HAGUE		Date of completion of the search 24 March 2000	Examiner Archontopoulos, E
<p>CATEGORY OF CITED DOCUMENTS</p> <p>X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document</p> <p>T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document</p>			

EPO FORM 1503 03.82 (P04C01)



European Patent
Office

EUROPEAN SEARCH REPORT

Application Number
EP 96 12 0614

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X	OBJECT MANAGEMENT GROUP: "The Common Object Request Broker: Architecture and Specification. Revision 2.0" July 1995 (1995-07), OBJECT MANAGEMENT GROUP INC. XP002133788	1-3,7, 11-14, 16-18, 21,22, 24,25 4,6 5,8,26	
Y	* page 2-2, line 1 - page 2-3, last paragraph *		
A	* page 2-6, paragraph 2.1.4 - page 2-9, paragraph 2.1.13 *		
	* page 2-10, paragraph 2.2.3 - page 2-12, line 4 *		
	* page 3-1, line 1 - line 8 *		
	* page 3-12, last paragraph - page 3-13, line 5 *		
	* page 3-27, paragraph 3.10 - page 3-28, paragraph 3.10.2 *		
	* page 4-3, line 3 - line 5 *		
	* page 14-8, paragraphs 14-7; table 19 *		
	* page 16-10, paragraph 16.5; table 23 *		
A	ANDREW D BIRRELL ET AL: "Implementing remote procedure calls" ACM TRANSACTIONS ON COMPUTER SYSTEMS, US, NEW YORK, NY, vol. 1, no. 2, 1 February 1984 (1984-02-01), pages 39-59, XP002074181 ISSN: 0734-2071 * page 43, paragraph 1.5 * * page 45, paragraph 2.1 * * page 46, line 25 - line 28 * * page 46, line 38 - line 43 *	1,2,7,8, 11-14, 16-18, 21,22, 24,26	
The present search report has been drawn up for all claims			
Place of search THE HAGUE		Date of completion of the search 24 March 2000	Examiner Archontopoulos, E
CATEGORY OF CITED DOCUMENTS X: particularly relevant if taken alone Y: particularly relevant if combined with another document of the same category A: technological background O: non-written disclosure P: intermediate document		T: theory or principle underlying the invention E: earlier patent document, but published on, or after the filing date D: document cited in the application L: document cited for other reasons &: member of the same patent family, corresponding document	

EPO FORM 1503 03.02 (P04021)



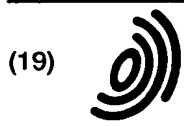
European Patent
Office

EUROPEAN SEARCH REPORT

Application Number
EP 96 12 0614

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
A	NARENDER V R ET AL: "DYNAMIC RPC FOR EXTENSIBILITY" PROCEEDINGS OF THE ANNUAL INTERNATIONAL PHOENIX CONFERENCE ON COMPUTERS AND COMMUNICATIONS,US,NEW YORK, IEEE, vol. CONF. 11, 1992, pages 93-100, XP000310598 ISBN: 0-7803-0605-8 * page 94, right-hand column, paragraph IV - page 96, left-hand column, paragraph VIII *	1,12,16, 21,24	
A	"LOCAL REMOTE PROCEDURE CALL EXTENSIONS FOR DISTRIBUTED COMPUTER ENVIRONMENT" IBM TECHNICAL DISCLOSURE BULLETIN,US,IBM CORP. NEW YORK, vol. 37, no. 12, 1 December 1994 (1994-12-01), pages 473-474, XP000487856 ISSN: 0018-8689 * the whole document *	26	
The present search report has been drawn up for all claims			TECHNICAL FIELDS SEARCHED (Int.Cl.6)
Place of search	Date of completion of the search	Examiner	
THE HAGUE	24 March 2000	Archontopoulos, E	
CATEGORY OF CITED DOCUMENTS		T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document	
X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document			

EPO FORM 1503 03/82 (P4/C01)



(19)

Europäisches Patentamt

European Patent Office

Office européen des brevets



(11)

EP 0 772 368 A2

(12)

EUROPEAN PATENT APPLICATION

(43) Date of publication:

07.05.1997 Bulletin 1997/19

(51) Int. Cl.⁶: **H04Q 11/04**

(21) Application number: **96117003.2**

(22) Date of filing: **23.10.1996**

(84) Designated Contracting States:
DE FR GB NL SE

(30) Priority: **02.11.1995 US 552342**

(71) Applicant: **SUN MICROSYSTEMS, INC.**
Mountain View, CA 94043 (US)

(72) Inventors:
• **Gentry, Denny E.**
Palo Alto, California 94306 (US)

• **Oskouy, Rasoul M.**
Fremont, California 94539 (US)

(74) Representative: **Schmidt, Steffen J., Dipl.-Ing.**
Wuesthoff & Wuesthoff,
Patent- und Rechtsanwälte,
Schweigerstrasse 2
81541 München (DE)

(54) **Method and apparatus for burst transferring ATM packet header and data to a host computer system**

(57) A network interface circuit (NIC) is provided with logic for maintaining various control pointers and at least one control counter for controlling burst transferring of buffered ATM cells to its host computer system in a non-cell-boundary-aligned block manner, distinguishing the ATM packet header from the ATM data most of the time, except for a number of predetermined exceptions. More specifically, ATM packet headers and ATM data are to be burst transferred to separate header and data buffers on the host computer system, except for short and atypical packets, in fixed size blocks, where the block size is complementary to the interface bus, but not necessarily aligned with the ATM cell boundaries. For the short and atypical packets, both the header and data are to be burst transferred into the header buffer instead. The logic employs a two phase approach to determining the appropriate updates to the relevant control pointers and at least one control counter after each burst transfer of header/data to the header/data buffer. In one embodiment, the logic is provided to the lookahead state machine of an unload block, which is part of the receive block of a system and ATM layer core of the NIC.

EP 0 772 368 A2

Description

BACKGROUND OF THE INVENTION

1. Field of the invention

The present invention relates to the field of computer networking. More specifically, the present invention relates to the transfer of asynchronous transfer mode (ATM) packet header and ATM data from a network interface circuit (NIC) to its host computer system.

2. Background Information

In U.S. Patent Application, S/N: 08/473,514, the claimed subject matter of which was invented by the same inventors, and assigned to the same assignee as the present invention, a method and apparatus for reordering interleaved ATM cells is disclosed. Described in U.S. Patent Application S/N: 08/473,514, interleaved ATM cells of different channels received by a NIC are buffered in "buckets" linked together as multiple linked lists on a per channel basis. The NIC would burst transfer the buffered ATM cells for a channel to the host computer system after the number of accumulated ATM cells for the channel has reached a predetermined threshold. A number of data structures and a schedule queue are used to manage the buffering of the ATM cells, including the location of the free resources, and scheduling the transfers. A method for burst transferring the buffered ATM cells (without distinguishing packet header and data) in fixed size blocks, where the block size is complementary to the interface bus between the NIC and its host computer system, but not necessarily aligned with the ATM cell boundaries, was also disclosed. Under the disclosed method, a partial bucket pointer in conjunction with a partial offset are employed for each channel to manage the temporal existence of a partially unloaded ATM bucket from time to time in the course of unloading. The partial bucket pointer is used to identify the partially unloaded ATM bucket, whereas the partial offset is used to identify the starting location of the residual data within the partially unloaded ATM bucket.

It is further desirable to be able to distinguish the packet header and data most of the time, when burst transferring ATM cells in the above described block manner. More specifically, it is desirable to be able to burst transfer the ATM packet headers and the ATM data into different buffers on the host computer system, except for short packets and "atypical" packets, in the above described block manner. A short packet is a packet with only a few bytes of data following a relatively lengthy header, whereas an "atypical" packet is a packet smaller than the header size of the most common packet type for which the hardware is programmed to optimize¹. For each of these packets, for performance reasons, both the packet header and data are to be burst transferred into the header buffer. As will be dis-

closed in more detail below, the present invention achieves these and other desired.

SUMMARY OF THE INVENTION

A NIC is provided with logic for maintaining various control pointers and at least one control counter for controlling burst transferring of buffered ATM cells to its host computer system in a non-cell-boundary-aligned block manner, distinguishing the ATM packet header from the ATM data most of the time, except for a number of predetermined exceptions. More specifically, ATM packet headers and ATM data are to be burst transferred to separate header and data buffers on the host computer system, except for short and atypical packets, in fixed size blocks, where the block size is complementary to the interface bus, but not necessarily aligned with the ATM cell boundaries. For the short and atypical packets, both the header and data are to be burst transferred into the header buffer instead.

The relevant control pointers include the partial bucket pointer and the partial offset. Additionally, the relevant control pointers include a first bucket pointer pointing to the first full ATM bucket, which follows the partial bucket pointer, if the partial bucket exists, a next bucket pointer pointing to the ATM bucket immediately following the first ATM bucket, and a last bucket pointer pointing to the last linked ATM bucket. The relevant at least one control counter includes a remaining header length counter.

The logic employs a two phase approach to determining the appropriate updates to these relevant control pointers and at least one control counter after each burst transfer of header/data to the header/data buffer. The logic takes into account whether the NIC is burst transferring the header or burst transferring the data, and whether the NIC is about to transition from burst transferring the header to burst transferring the data, when determining the appropriate updates. Furthermore, the logic takes into account whether the packets are to be handled as exceptions, i.e. not distinguishing the data from the header.

In the first phase, the logic determines the new remaining header length, if the NIC is burst transferring the header, and the new "unnormalized" partial offset. In the second phase, the logic determines the new "normalized" partial offset, the new partial, first and next bucket pointers, depending on where the new "unnormalized" partial offset is pointing to. More specifically, the logic makes the above determinations depending on whether the new "unnormalized" partial offset is pointing to a position in the linked ATM buckets that is beyond

¹ Applications employ these atypical packets for control purpose. For example, the most common packet type has a header length of 206 bytes, however a control packet has a header length of 36 bytes and data length of 4 bytes totalling to only 40 bytes.

the furthest full ATM cell that could have been burst in the just completed burst (hereinafter simply the "furthest full ATM bucket", at the end of the "furthest full ATM bucket", beyond the ATM bucket where the just completed burst started (hereinafter simply the "starting ATM bucket") but before the end of the "furthest full ATM bucket", at the end of the "starting ATM bucket", or still within the "starting ATM bucket". For each of these cases, the logic further factors into consideration whether the EOP marking was among the header/data that was just burst transferred.

In one embodiment, the logic is provided to the lookahead state machine of an unload block, which is part of the receive block of a system and ATM layer core of the NIC.

BRIEF DESCRIPTION OF DRAWINGS

The present invention will be described by way of exemplary embodiments, but not limitations, illustrated in the accompanying drawings in which like references denote similar elements, and in which:

Figure 1 illustrates an exemplary network of computer systems incorporating the teachings of the present invention;

Figure 2 illustrates one embodiment of the NIC of **Fig. 1**;

Figure 3 illustrates the system and ATM layer core of **Fig. 2**;

Figure 4 illustrates the Receive Block of **Fig. 3**;

Figure 5 illustrates an ATM packet;

Figure 6 illustrates the header and data buffers on the host system;

Figures 7 - 8 illustrate the non-cell-boundary-aligned approach for burst transferring header/data to the host system;

Figure 9 illustrates one embodiment of the Unload Block of **Fig. 4**; and

Figure 10a - 10f illustrate one embodiment of the logic flow incorporated in the lookahead state machine of **Fig. 9** for maintaining the relevant control pointers and counter(s) for burst transferring ATM header and data in the desired manner.

DETAILED DESCRIPTION OF THE INVENTION

In the following description, for purposes of explanation, specific numbers, materials and configurations are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without the specific details. In other instances, well known features are omitted or simplified in order not to obscure the present invention.

Figure 1 illustrates an exemplary computer system network incorporating the ATM NIC of the present invention. Computer system network 10 includes host computer systems (not shown) which incorporate one

or more ATM NIC 12. NICs 12 are coupled through local ATM switches 14 to public ATM switches 16 to enable asynchronous transfer of data between host computer systems coupled to network 10. Alternatively, NICs 12 can be coupled directly to public ATM switches 16. As shown in **Figure 1**, computer system network 10 may also include computer systems which incorporate the use of a Local Area Network (LAN) emulation 15, which serves as a gateway for connecting other networks such as Ethernet or token ring networks 17 which utilize the ATM network as a supporting framework.

Figure 2 is a simplified system diagram illustrating the architecture of ATM NIC 12 in accordance with a preferred embodiment of the present invention. ATM NIC 12 interfaces host computer system 48 coupled through system bus 38 to network ATM cell interface 40 operating in accordance with the ATM protocol.

ATM NIC 12 shown includes System Bus interface 20, Generic Input/Output (GIO) interface 24, System and ATM Layer Core 22, Local Slave interface 26, transmit (TX) FIFO 28, receive (RX) FIFO 30, Cell Interface Block 32, External Buffer Memory interface 34 and clock synthesis circuit 36.

Together, elements 20 - 36 of NIC 12 cooperate to transfer data between host computer 48 and the other computers in the network through multiple, dynamically allocated channels in multiple bandwidth groups. Collectively, the elements of NIC 12 function as a multi-channel intelligent direct memory access (DMA) controller coupled to System Bus 38 of host computer system 48. In a preferred embodiment, multiple transmit and receive channels function as virtual connections utilizing a full duplex 155/622 Mbps physical link. Multiple packets of data, subscribed to different channels over System Bus 38 to external buffer memory 42, via External Buffer Memory interface 34, are segmented by System and ATM Layer Core 22 into transmit cells for transmission to ATM cell interface 40 through Cell Interface Block 32. Core 22 includes reassembly logic to facilitate reassembly of the received cells to packets.

Three memory sub-systems are associated with the operation of the NIC 12. These include host memory 49 located in host computer system 48, external buffer memory 42 external to NIC 12 and storage block 44 located in the Core 22. NIC 12 manages two memory areas: external buffer memory 42 and storage block 44. External buffer memory 42 contains packet data for all transmit and receive channels supported by NIC 12. Storage block 44 contains DMA state information for transmit and receive channels and pointers to data structures in host memory 49 for which DMA transfers are performed. Storage block 44 also contains the data structure specifics to manage multiple transmit and receive buffers for packets in transition between host 48 and ATM Cell Interface 40.

Host computer system 48 includes host memory 49 which contains data packets and pointers to the packets being transmitted and received. As noted previously, NIC 12 also shields the cell delineation details of asyn-

chronous transfer from the applications on host computer system 48. For present purposes, it is assumed that software running on host computer system 48 transmits and receive data using wrap around transmit and receive rings with packet interfaces as is well known in the art.

TX and RX FIFOS 28,30, coupled between Core 22, and Cell Interface Block 32, are used to stage the transmit and receive cell payloads of the transmit and receive packets respectively. The Cell Interface Block 32 transmits and receives cells to the ATM Cell Interface 40 of the network, driven by clock signals provided by Clock Synthesis Circuit 36. Preferably, ATM Cell Interface 40, and therefore ATM Cell Interface 32, conforms to the Universal Test and Operations Physical Interface for ATM (UTOPIA) standard, as described by the ATM Forum Ad Hoc specification. To conform to the UTOPIA specification, the clock synthesis circuit 36 provides either a clock signal of 20-25 MHz or 40-50 MHz to enable Cell Interface Block 32 to support an 8-bit stream of 20-25 MHz for 155 Mbps or a 16-bit stream at 40-50 MHz for a 622 Mbps data stream.

In the presently preferred embodiment, Cell Interface Block 32 receives 52 byte data cells each having a 4 byte cell header and a 48 byte payload from TX buffer memory 46 through TX FIFO 28 under the control of Core 22, in groups of 4 bytes. Cell Interface Block 32 inserts a checksum as a fifth byte to the cell header into each cell prior to providing the 53 byte data cell to ATM Cell Interface 40 at either 155 or 622 Mbps. Conversely, when Cell Interface Block 32 receives cells from ATM Cell Interface 40, it examines the checksum in the fifth byte of each cell to determine if the checksum is correct. If so, the byte representing the checksum is stripped from the cell and the cell is forwarded to RX FIFO 30 4 bytes at a time at either 155 or 622 Mbps. Otherwise, the entire cell is dropped. Transferred bytes are stored in RX buffer memory 45 via external Buffer Memory Interface 34 under the control of Core 22.

In one embodiment, TX and RX FIFOS 28 and 30 are 33 bits wide, of which 32 bits are used for transmitting data and 1 bit is used as a tag. The tag bit is used to differentiate the 4-byte cell header from the 48-byte cell payload. The tag bit is generated by TX block 50 located within Core 22. In one embodiment, the tag bit is set to 1 to indicate the start of a cell header and the tag bit is reset to 0 to indicate a cell payload. Thus, the tag bit is 1 for the first 4 bytes of the cell (header) and then the tag bit is 0 for the remaining 48 bytes of the cell (cell payload).

Upon receiving the data cells from TX FIFO 28, TX circuit 53 located within Cell Interface block 32 examines the tag bit. If the tag bit is a 1, TX circuit 53 decodes the corresponding 32 bits as the header of the cell. If the tag bit is 0, TX circuit 53 decodes the corresponding 32 bits as data. Conversely, when Cell Interface block 32 receives data cells from ATM Cell Interface 40, RX block 55 in Cell Interface block 32 generates a tag bit to differentiate the 4-byte cell header from the 48-byte cell pay-

load. Cell Interface block 32 then dispatches the data cells in groups of 4 bytes to RX FIFO 30. Upon receipt of the cell data from RX FIFO 30, RX circuit 52 in the Core 22 decodes the cell data in accordance with the value of the tag bit as discussed above.

Two synchronous clock signals, a 20 MHz signal and a 40 MHz signal, are provided to Cell Interface block 32 from the ATM Cell Interface Clock via the Clock Synthesis circuit 36. A 40 MHz clock is supplied to provide a 16-bit data stream at 40 MHz for 622 Mbps in accordance with the specifications of UTOPIA. A divide by 2 of the 40 MHz clock signal is performed in the Clock Synthesis circuit 36 to provide an 8-bit data stream at 20 MHz for 155 Mbps in accordance with the specifications of UTOPIA. The 40 MHz clock signal is also provided to the external buffer memory interface 34 for providing a 1.2 Gbps transmission rate. In addition, GIO 24 uses the 40 MHz clock signal for transmitting and receiving data.

TX Buffer Memory 46 provides 32 bits of data to the TX FIFO 28 and RX Buffer Memory 45 reads 32 bits of data from RX FIFO 30 at every cycle of the 40 MHz clock signal. However, ATM Cell Interface 40 reads 4 bytes of data from TX FIFO 28 every two clock cycles when operating at 622 Mbps, and reads 4 bytes of data from TX FIFO 28 every 8 clock cycles when operating at 155 Mbps. In the same manner, Cell Interface block 32 provides 4 bytes of data to TX FIFO 28 every two clock cycles when operating at 622 Mbps, and provides 4 bytes of data to TX FIFO 28 every 8 clock cycles when operating at 155 Mbps. Although the cell burst rate of Core 22 is different from the cell burst rate of Cell Interface block 32, the data rate between TX FIFO 28 and Cell Interface block 32 is, on average, the same as the data rate between the between TX FIFO 28 and Core 22. Similarly, the data rate between RX FIFO 30 and Cell Interface block 32 is on average, the same as the data rate between the RX FIFO 28 and Core 22. This is because the data rate between TX and RX FIFOS 28 and 30 and Core 22 is dependent the rate that data is read or written by Cell Interface block 32 respectively. In one embodiment, the depth of TX FIFO 28 is 18 words or 1 1/2 cells long and the depth of RX FIFO 30 is 70 words long.

System Bus Interface 20 and GIO interface 24 insulate host computer system 48 from the specifics of the transfer to ATM Cell Interface 40. Furthermore, Core 22 is insulated from the specifics of system bus 38 and host specifics. In the presently preferred embodiment, the system bus is an S-Bus, as specified in the Institute of Electronics and Electrical Engineers (IEEE) standard 1496 specification. System Bus interface 20 is configured to communicate in accordance with the specifications of the system bus, in the present illustration, the S-Bus. It is contemplated that System Bus Interface 20 can be configured to conform to different host computer system busses. System Bus interface 20 is also configured to transfer and receive data in accordance with the protocols specified by GIO interface 24. GIO interface

24 provides a singular interface through which Core 22 communicates with the host computer. Thus, Core 22 does not change for different embodiments of NIC 12 which interface to different host computer systems and busses.

Figure 3 illustrates Core 22 in further detail. As shown, Core 22 comprises TX block 50, RX block 52, arbiter 54 and control memory 56 coupled to each other as shown. TX block 50 is used to receive ATM packets from the host computer system, segment them into ATM cells and provide the segmented ATM cells to TX FIFO 28. In the presently preferred embodiment, TX block 50 incorporates teachings of copending US Patent Application, Application No: x/xxx,xxx, allowing TX block 50 to segment ATM packets at 622 Mbps or higher. RX block 52 is used to receive ATM cells from RX FIFO 30, reassemble them into ATM packets and provide the reassembled ATM packets to the host computer system. RX block 52 incorporates the teachings disclosed in the above identified copending US Patent Application for reordering the interleaved ATM cells for the different channels. In other words, received ATM cells for various channels are stored in ATM buckets in external memory 42. The ATM buckets are organized into linked lists on a per channel basis. The unused or empty ATM buckets are organized into a free resource linked list. Furthermore, as will be described in more detail below, RX block 52 incorporates the teachings of the present invention for burst transferring buffered ATM cells to the host computer system in a block manner, where the transfer blocks are not necessarily aligned with ATM cell boundaries, and at the same time, distinguishing ATM packet headers and data, except for a number of predetermined exceptions, i.e. the short packets and the atypical packets described earlier.

Control RAM 56 is used to store various control information for TX and RX blocks 50 and 52 including in particular, the linkage information for the above described ATM bucket linked lists, the relevant control pointers for unloading the ATM cells in the desired manner, and at least one control counter to be described more fully below. Lastly, arbiter 54 is used to arbitrate accesses to control RAM 56 between TX and RX blocks 50 and 52. Control RAM 56 and arbiter 54 may be implemented in a number of manners well known in the art.

Figure 4 illustrates one embodiment of RX block 52 in further detail. As shown, for the illustrated embodiment, RX block 52 comprises RX Load block 58, RX Unload block 60, schedule queue 62, free list manager 64, external RAM interface 66, and control RAM interface 68, coupled to each other as shown. RX Load block 58 is used to receive the interleaved ATM cells from RX FIFO 30, and stored them in free ATM buckets in external memory 34. Free list manager 64 is used to manage the free resources, informing RX Load block 58 where the free buckets are located. RX Load block 58 is also used to schedule channels requiring unload services with schedule queue 62. In one embodiment, unload

service is scheduled whenever the number of ATM cells accumulated for a channel reaches a predetermined threshold. RX Unload block 60 is used to unload the buffered ATM cells, and burst transfer them to the host computer system in the desired manner, i.e. in fixed size block where the block size is complementary to the bus interface, and yet at the same time distinguishing ATM packet header and data, except for the enumerated predetermined exceptions. For further description on the fundamental operations of RX Load block 58, RX Unload block 60, schedule queue 62 and free list manager 64, see copending US Patent Application, S/N: 08/473,514.

External and Control RAM interfaces 66 and 68 perform their conventional interfacing functions to the respective memories. External and Control RAM interfaces 66 and 68 may also be implemented in any number of approaches well known in the art.

Before describing RX Unload block 60 in further detail, we will first describe the desired manner of transfer, referencing Figures 5-8. Figure 5 illustrates a "typical" ATM packet 70 comprising a header 72, data 74 and EOP 76. As shown, ATM packet 70 is received in multiple ATM cells (packaged with cell headers) 78. Furthermore, the ATM cells of different channels arrived interleavably. In other words, ATM cells 78 of ATM packet 70 are not necessarily received into RX FIFO 30 successively. As described earlier, however, some packets are relatively short, i.e., header 72 is relatively lengthy and data 74 is only a few bytes long. Furthermore, there are also atypical packets, wherein the entire packet is shorter than the header of the most common packet type.

Figure 6 illustrate the desired manner packet headers and packet data are placed into the host computer system, i.e. the packet header is placed into header buffer 80, and the packet data are placed into data buffer 82, which is separate from header buffer 80, except for the short and atypical packets. For these packets, both the packet header and packet data are to be placed into header buffer 80.

Figures 7 - 8 illustrate the desired manner of transferring ATM header 72 and ATM data 74 to their respective buffers 80 and 82. As shown, ATM cells 78 stored in the ATM buckets. The ATM buckets are linked together as linked lists on a per channel basis (linkage information not shown). Additionally, each linked list is managed with a number of pointers. More specifically, each linked list is managed with a first bucket pointer 84 pointing to the first full bucket (as opposed to a partial buckets or a bucket with residual data), a next bucket pointer 86 pointing to the next ATM bucket immediately following the first bucket, and a last bucket pointer 88 pointing to the last ATM bucket.

Stored ATM cells are transferred to the host computer system in fixed sized blocks, where the block size is complementary to the bus interface and not equal to the ATM cell size. Thus, it is possible for partially unloaded bucket or bucket with residual data to exist from time to

time. (Note that there is only one partial bucket at a time for each channel.) Therefore, in addition to the above described pointers 84 - 88, a partial bucket pointer 90 is also employed for each channel to identify the partial bucket, and a partial offset 92 is used to identify the starting location of the residual data within the partial bucket. Furthermore, until the header is completely transferred, it is necessary to maintain the remaining header length, thus remaining header length counter 94 is employed for that purpose.

Control pointers 84 - 92 and control counter 94 are updated after each burst transfer. More specifically, first pointer 84, next pointer 86, partial pointer 92 and partial offset 94 are all "advanced" accordingly, and remaining header length 94 are decremented accordingly. Last bucket pointer 88 is updated when all buffered ATM cells for the channel have been transferred to the host computer system.

Since partial offset is used to denote the starting location of the residual data in a partial ATM bucket, this partial offset can take on values that are between 0 and the ATM cell size (CS) only. Partial offset is equal to 0 when there is no partial bucket. On the other hand partial offset is equal to CS if it is pointing to the end of the ATM bucket. Thus, to determine the correct new partial offset after each burst transfer, upon nominally incrementing the partial offset by the burst size (BS), it must be adjusted or normalized so that the partial offset value falls between 0 and CS.

Additionally, since CS and the BS are fixed, the values incremented partial offset can take on prior to normalization are finite and predeterminable. For example, in the presently preferred embodiment, CS is 48 bytes, and BS is 64 bytes. Thus, the incremented partial offset value prior to normalization is necessarily between 0 and 28 (in units of words). As a further illustration, if BS is changed to 128 bytes, the incremented partial offset value prior to normalization is necessarily between 0 and 44 (in units of words).

Furthermore, for the purpose of managing the bucket pointers and the partial offset, the pre-normalization incremented partial offset value (p) can be analyzed as a finite number of predeterminable cases. Consider the presently preferred embodiment again (CS = 48 bytes and BS = 64 bytes), since p is necessarily between 0 - 28 (in units of words), the stopping location of the just completed burst transfer is beyond the "furthest full ATM bucket" if p is greater than 24, within the "furthest full ATM bucket"² if p is between 13 - 24, and within the "starting bucket" when p is between 0 - 12. As a further illustration, for the embodiment where CS is 48 bytes and BS is 128 bytes, since p is necessarily between 0 - 44 (in units of words), the stopping location of the just completed burst transfer is beyond the "furthest full ATM bucket" if p is greater than 36, within the "furthest full ATM bucket" if p is between 25 - 36, within

the bucket immediately following the "starting bucket" if p is between 13 - 24, and within the "starting bucket" when p is between 0 - 12.

Having now described the desired manner of transfer, we will now describe RX Unload block 60 in further detail, in particular, the teachings of the present invention incorporated therein, referencing Figures 9 and 10a - 10f. Figure 9 illustrates one embodiment of RX Unload block 60. As shown, RX Unload block 60 comprises data engine state machine 96, lookahead state machine 98, a get buffer state machine 100, a number of unload registers 102 and a number of adders, subtractors, comparators and multiplexors 104, coupled to each other as shown. Get buffer state machine 100 is used to get buffers on the host computer system, whereas data engine state machine 96 is used to actually burst transfer the header and data to the appropriate buffers on the host computer system. Lookahead state machine 98 is used to control the operation of RX unload block 60. Unload registers 102 are used by the various state machines 96 - 100, in particular, by lookahead state machine 98, for storing various control data. Modifications to the control data are accomplished by reading the control data out of unload registers 102, providing the read out data to selected ones of the adders, subtractors, etc. 104, and then operating on the provided control data using selected ones of the adders, subtractors etc. 104.

Lookahead state machine 98 provides the appropriate control information and control signals to get buffer state machine 100 and data engine state machine 96, including the above described first bucket pointer 84, next bucket pointer 86, etc. Lookahead state machine 98 monitors the unloading being performed, including whether it is header burst transfer or data burst transfer that is being performed and whether an EOP was detected among the header/data that was just burst transferred. In turn, lookahead state machine 98 maintains the control data accordingly, using unload registers 102 and the adders, subtractors etc. 104.

The manner in which lookahead state machine 98 causes the relevant control pointers 84 - 92 and the at least one relevant counter 94 to be properly maintained, thereby allowing the ATM cells to be burst transferred to the host computer system in the desired manner, will be described in more detail below. The other functions performed by get buffer state machine 100, data engine state machine 96, as well as lookahead state machine 98, are not directly relevant with respect to understanding the present invention. Thus, they will not be further described.

Figures 10a - 10f illustrate one embodiment of the operational logic of lookahead state machine 98 for maintaining the various relevant control pointers 84 - 92 and the remaining header length counter 94. As shown in Fig. 10a, lookahead state machine 98 first determines if the burst transfer is a header burst transfer or a data burst transfer, step 200. If the burst transfer is determined to be a header burst transfer, lookahead

² The "furthest full ATM bucket" in this case is the bucket immediately following the "starting bucket".

state machine 98 causes a new remaining header length to be computed by subtracting the burst size from the old remaining header length, step 202. Next, lookahead state machine 98 determines if the new header length is less than zero, step 204. If new header length is determined to be less than zero, lookahead state machine 98 causes new partial offset (unnormalized) 92 to be computed, by adding the old header length 94 to the old partial offset 92, step 206. Furthermore, lookahead state machine 98 causes the new header length 92 to be adjusted to zero, step 208.

On the other hand, if it was determined at step 200 that the burst transfer is a data burst transfer, or at step 204 that the new header length 92 is not less than zero, lookahead state machine 98 simply causes a new partial offset 92 to be computed by adding the burst size to the old partial offset 92, step 210.

Next, lookahead state machine 98 determines if the pre-normalization new partial offset 92 is pointing at a location beyond the "furthest full ATM bucket", step 212. If the determination is positive, lookahead state machine 98 causes the relevant pointers 84 - 92 to be updated in accordance to the "beyond furthest full ATM bucket" case, step 214. On the other hand, if the determination is negative, lookahead state machine 98 further determines if the prenormalization new partial offset is pointing precisely at the end of the "furthest full ATM bucket", step 216.

If the determination at step 216 is positive, lookahead state machine 98 causes the relevant pointers 84 - 92 to be updated in accordance to the "end of furthest full ATM bucket" case, step 218. On the other hand, if the determination at step 216 is negative, lookahead state machine 98 further determines if the prenormalization new partial offset is pointing beyond the "starting bucket", step 220.

Again, if the determination at step 220 is positive, lookahead state machine 98 causes the relevant pointers 84 - 92 to be updated in accordance to the "beyond the starting bucket" case, step 222. On the other hand, if the determination at step 220 is negative, lookahead state machine 98 further determines if the pre-normalization new partial offset is pointing precisely at the end of the "starting bucket", step 224.

Likewise, if the determination at step 224 is positive, lookahead state machine 98 causes the relevant pointers 84 - 92 to be updated in accordance to the "end of starting bucket" case, step 226. On the other hand, if the determination at step 224 is negative, lookahead state machine 98 causes the relevant pointers 84 - 92 to be updated in accordance to the "within starting bucket" case, step 228.

Figures 10b - 10f illustrate how the relevant pointers 84 - 92 are updated under the above described cases. For ease of explanation, the updates will be illustrated in accordance to the presently preferred embodiment where CS is 48 bytes and BS is 64 bytes. In other words, under the presently preferred embodiment, as described earlier, the valid values (in units of words)

partial offset 92 can take on are 0 - 12. The largest value (in units of words) partial offset 92 can take on before a burst transfer is 12, and the largest value (in units of words) pre-normalization new partial offset 92 can take on is 28 (12 + 16, where 16 is BS in words). Therefore, for this embodiment, the appropriate adjustment values (in units of words) for normalizing the partial offset 92, are 24 (or $2 \times \text{CS}$) for the cases "beyond" and "at the end" of the "furthest full ATM bucket", and 12 ($1 \times \text{CS}$) for the cases "beyond" and "at the end" of the "starting bucket" cases.

Thus, as shown in Figure 10b, for the "beyond furthest full ATM bucket" case, at step 230, lookahead state machine 98 causes the normalized new partial offset 92 to be computed by subtracting $2 \times \text{CS}$ from the prenormalization new partial offset 92. Then, lookahead state machine 98 causes the former partial bucket and the former first bucket to be returned to the free resource list, step 232. Next, lookahead state machine 98 determines if EOP was detected in the previous first full ATM bucket, i.e. the bucket pointed to by the first bucket pointer prior to the just completed burst transfer, step 234. If the determination is positive, lookahead state machine 98 would set the partial bucket pointer 90 to be equal to the previous next bucket pointer 86, and look up the appropriate address value for new first bucket pointer 84 using the linkage information maintained. On the other hand, if the determination is negative, lookahead state machine 98 sets new partial bucket pointer 90 to null, normalized new partial offset 92 to zero, and the first bucket pointer 84 to equal to the previous next bucket pointer 86, step 238.

As shown in Figure 10c, for the "at the end of the furthest full ATM bucket" case, at step 240, lookahead state machine 98 causes the normalized new partial offset 92 to be computed by subtracting $2 \times \text{CS}$ from the prenormalization new partial offset pointer 92. Then, lookahead state machine 98 causes the former partial bucket and the first bucket to be returned to the free resource list, step 242. Furthermore, lookahead state machine 98 sets new partial bucket pointer 90 to null, and the first bucket pointer 84 to equal to the previous next bucket pointer 86, step 244. (Note that the partial offset 92 was effectively set to zero at step 240.)

As shown in Figure 10d, for the "beyond starting ATM bucket" case, at step 246, lookahead state machine 98 causes the normalized new partial offset 92 to be computed by subtracting $1 \times \text{CS}$ from the prenormalization new partial offset 92. Then, lookahead state machine 98 determines if the "starting bucket" was a partial bucket, step 248. If the "starting bucket" was a partial bucket, lookahead state machine 98 causes the "starting partial bucket" to be returned to the free list, step 250. Then, lookahead state machine 98 sets the partial bucket pointer 90 to equal to the previous first bucket pointer 84, and the first bucket pointer 84 to equal the previous next bucket pointer 86.

However, if it was determined back at step 248, that the "starting bucket" was not a partial bucket, lookahead

state machine 98 causes the "starting bucket" to be returned to the free list, step 254. Next, lookahead state machine 98 determines if EOP was detected at the former first bucket, step 256. If the determination is positive, lookahead state machine 98 sets partial pointer 90 equal to previous next pointer 86, and looks up the address value for new first pointer 84 using the linkage information maintained. On the other hand, if the determination is negative, lookahead state machine 98 sets partial offset 92 to zero, and set first pointer 84 to equal to previous next pointer 86, step 260.

As shown in Figure 10e, for the "at the end of starting ATM bucket" case, at step 262, lookahead state machine 98 causes the normalized new partial offset 92 to be computed by subtracting 1xCS from the prenormalization new partial offset 92. Then, lookahead state machine 98 determines if the "starting bucket" was a partial bucket, step 264. If the "starting bucket" was a partial bucket, lookahead state machine 98 causes the "starting partial bucket" to be returned to the free list, step 266. Then, lookahead state machine 98 sets the partial bucket pointer 90 to equal to zero, and the first bucket pointer 84 to equal the previous next bucket pointer 86. On the other hand, if back at step 264, it was determined lookahead state machine 98 causes the starting first bucket to be returned to the free list, step 270. Then, lookahead state machine 98 sets the first bucket pointer 84 to equal the previous next bucket pointer 86.

Lastly, as shown in Figure 10f, for the "within starting bucket" case, lookahead state machine 98 determines if the "starting bucket" was a partial bucket, step 274. If the determination is positive, no further action is taken, i.e. all pointers as well as the partial offset are to remain unchanged. On the other hand, if the determination was negative, lookahead state machine 98 sets the partial bucket pointer 90 to equal to the previous first pointer 84, and the first pointer to equal the previous next pointer 86, step 276.

Thus, a method and apparatus for burst transferring ATM packet header and data to a host computer system has been described. While the method and apparatus of the present invention has been described in terms of the above illustrated embodiments, those skilled in the art will recognize that the invention is not limited to the embodiments described. The present invention can be practiced with modification and alteration within the spirit and scope of the appended claims. The description is thus to be regarded as illustrative instead of restrictive on the present invention.

Claims

1. An apparatus comprising:

(a) a memory unit for storing a plurality of control data for a plurality of linked storage buckets for storing a plurality of asynchronous transfer mode (ATM) cells of a plurality of packets for a

plurality of channels, each packet having a packet header and packet data; and

(b) a receive block coupled to the memory unit for managing receipt of the ATM cells into the linked storage buckets, and burst transferring the packet headers and the packet data out of the linked storage buckets into separate header and data buffers of the various channels on a host computer system coupled to the apparatus through an interface bus, except for a number of predetermined exceptions under which both packet header and data are burst transferred into the header buffers, the packet headers and data being burst transferred in fixed size blocks, each block have a block size that is complementary to the interface bus, but not necessarily aligned with cell boundaries, the receive block comprising logic for maintaining the control data, taking into account whether each burst transfer is a header burst transfer or a data burst transfer, if the burst transfer is a header burst transfer, whether the entire packet header has been completely transferred at the end of the particular burst transfer, and additionally, whether the packet is to be handled exceptionally.

2. The apparatus as set forth in claim 1, wherein the logic is disposed in an unload block of the receive block, the unload block comprising

(b.1) a plurality of registers for storing a working copy of the control data of a channel whose packet header or data is being burst transferred to the host computer system,

(b.2) a plurality of operation circuitry coupled to the registers for selectively performing numerical and logic operations on selected ones of the control data; and

(b.3) a state machine coupled to the operation circuitry and the registers for controlling the selective performance of the numerical and logical operations on selected ones of the control data to effectuate maintenance of the control data, taking into account whether each burst transfer is a header burst transfer or a data burst transfer, if the burst transfer is a header burst transfer, whether the entire packet header has been completely transferred at the end of the particular burst transfer, and additionally, whether the packet is to be handled exceptionally.

3. The apparatus as set forth in claim 1, wherein

the control data include for each channel a counter for tracking the length of the header remains to be burst transferred for the packet being burst transferred;

for each burst transfer, the logic also uses the zero and non-zero states of the remaining header length counter to determine whether the burst transfer is a header burst transfer or a data burst transfer; and

for each header burst transfer, the logic decrements the remaining header length counter by the block size, but not beyond zero.

4. The apparatus as set forth in claim 3, wherein

the control data further include for each channel a normalized partial offset for identifying a location within a partially unloaded storage bucket of the channel where residual header/data starts, if the partially unloaded storage bucket exists; and

for each burst transfer, the logic computes an unnormalized partial offset by incrementing the normalized partial offset by either the block size or the pre-burst transfer value of the remaining header length counter, depending on whether the burst transfer is a header burst transfer or a data burst transfer, and if it is a header burst transfer, further depending on whether the remaining header length counter has been decremented to zero.

5. The apparatus as set forth in claim 4, wherein,

the logic logically divides potential values that could be assumed by the unnormalized partial offset after each burst transfer into a plurality of ranges, each range denoting a portion of the linked storage buckets; and

for each burst transfer, the logic further determines which portion of the linked storage buckets the unnormalized partial offset is pointing to by determining which range numerically encompasses the unnormalized partial offset.

6. The apparatus as set forth in claim 5, wherein,

the control data further include a partial bucket pointer for identifying the partially unloaded storage bucket for the channel, a first bucket pointer for identifying the first full storage bucket for the channel, and a next bucket pointer for identifying the storage bucket immediately following the first storage bucket for the channel;

for each burst transfer, the logic further conditionally returns the partially unloaded storage bucket, the first full storage bucket and the next storage bucket, and maintaining the partial, first and next bucket pointers, depending on which portion of the linked storage buckets the unnormalized partial offset is pointing to.

7. The apparatus as set forth in claim 6, wherein, the portions of the linked storage buckets include

a first portion that is beyond the storage bucket whose ATM cell is the furthest ATM cell that could have been completely burst transferred by the burst transfer; and

a second portion that is precisely at the end of the storage bucket whose ATM cell is the furthest ATM cell that could have been completely burst transferred by the burst transfer.

8. The apparatus as set forth in claim 7, wherein, the portions of the linked storage buckets further include a third portion that is beyond the storage bucket where the just completed burst transfer started, but before the second portion.

9. The apparatus as set forth in claim 7, wherein, the portions of the linked storage buckets further include

a third portion that is precisely at the end of the storage bucket where the just completed burst transfer started; and

a fourth portion that is within the storage bucket where the just completed burst transfer started.

10. In a computer system, a method for burst transferring headers and data of packets from a network interface circuit of the system to buffers on the system, the method comprising the steps of:

(a) storing a plurality of control data for a plurality of linked storage buckets for storing a plurality of asynchronous transfer mode (ATM) cells of the packets on a per channel basis in a memory unit; and

(b) managing receipt of the ATM cells into the linked storage buckets, and burst transferring the packet headers and the packet data out of the linked storage buckets into separate header and data buffers of the various channels through an interface bus, except for a number of predetermined exceptions under which both packet header and data are burst transferred into the header buffers, the packet headers and data being burst transferred in fixed size blocks, each block have a block size that is complementary to the interface bus, but not necessarily aligned with cell boundaries,

(c) maintaining the control data, taking into account whether each burst transfer is a header burst transfer or a data burst transfer, if the burst transfer is a header burst transfer, whether the entire packet header has been completely transferred at the end of the particular burst transfer, and additionally, whether the packet is to be handled exceptionally.

11. The method as set forth in claim 10, wherein

the control data include for each channel a counter for tracking the length of the header remains to be burst transferred for the packet being burst transferred;

step (c) includes, for each burst transfer, using the zero and non-zero states of the remaining header length counter to determine whether the burst transfer is a header burst transfer or a data burst transfer; and

step (c) further includes, for each header burst transfer, decrementing the remaining header length counter by the block size, but not beyond zero.

12. The method as set forth in claim 11, wherein

the control data further include for each channel a normalized partial offset for identifying a location within a partially unloaded storage bucket of the channel where residual header/data starts, if the partially unloaded storage bucket exists; and

step (c) includes, for each burst transfer, computing an unnormalized partial offset by incrementing the normalized partial offset by either the block size or the pre-burst transfer value of the remaining header length counter, depending on whether the burst transfer is a header burst transfer or a data burst transfer, and if it is a header burst transfer, further depending on whether the remaining header length counter has been decremented to zero.

13. The method as set forth in claim 12, wherein,

step (c) further includes logically dividing potential values that could be assumed by the unnormalized partial offset after each burst transfer into a plurality of ranges, each range denoting a portion of the linked storage buckets; and

step (c) further includes, for each burst transfer, determining which portion of the linked storage buckets the unnormalized partial offset is pointing to by determining which range numerically encompasses the unnormalized partial offset.

14. The method as set forth in claim 13, wherein,

the control data further include a partial bucket pointer for identifying the partially unloaded storage bucket for the channel, a first bucket pointer for identifying the first full storage bucket for the channel, and a next bucket pointer for identifying the storage bucket immediately following the first storage bucket for the channel;

step (c) further includes, for each burst transfer, conditionally returning the partially unloaded storage bucket, the first full storage bucket and the next storage bucket, and maintaining the partial, first and next bucket pointers, depending on which portion of the linked storage buckets the unnormalized partial offset is pointing to.

15. The method as set forth in claim 14, wherein, the portions of the linked storage buckets include

a first portion that is beyond the storage bucket whose ATM cell is the furthest ATM cell that could have been completely burst transferred by the burst transfer; and

a second portion that is precisely at the end of the storage bucket whose ATM cell is the furthest ATM cell that could have been completely burst transferred by the burst transfer.

16. The method as set forth in claim 15, wherein, the portions of the linked storage buckets further include a third portion that is beyond the storage bucket where the just completed burst transfer started, but before the second portion.

17. The method as set forth in claim 16, wherein, the portions of the linked storage buckets further include

a third portion that is precisely at the end of the storage bucket where the just completed burst transfer started; and

a fourth portion that is within the storage bucket where the just completed burst transfer started.

18. A computer system comprising:

(a) a first memory unit for storing a plurality of header and data buffers for storing a plurality of headers and data for a plurality of packets for a plurality of channels;

(b) a second memory unit for storing a plurality of linked storage buckets for storing a plurality of asynchronous transfer mode (ATM) cells of the packets;

(c) a third memory unit for storing a plurality of control data for the linked storage buckets; and

(d) a receive block coupled to the memory units for managing receipt of the ATM cells into the linked storage buckets, and burst transferring the headers and data of the packets out of the linked storage buckets into the header and data buffers through an interface bus coupling the first and second memory units, except for a number of predetermined exceptions under which both packet header and data are burst transferred into the header buffers, the packet

headers and data being burst transferred in fixed size blocks, each block have a block size that is complementary to the interface bus, but not necessarily aligned with cell boundaries, the receive block comprising logic for maintaining the control data, taking into account whether each burst transfer is a header burst transfer or a data burst transfer, if the burst transfer is a header burst transfer, whether the entire packet header has been completely transferred at the end of the particular burst transfer, and additionally, whether the packet is to be handled exceptionally.

19. The computer system as set forth in claim 18, wherein the logic is disposed in an unload block of the receive block, the unload block comprising

(d.1) a plurality of registers for storing a working copy of the control data of a channel whose packet header or data is being burst transferred to the host computer system,
 (d.2) a plurality of operation circuitry coupled to the registers for selectively performing numerical and logic operations on selected ones of the control data; and
 (d.3) a state machine coupled to the operation circuitry and the registers for controlling the selective performance of the numerical and logical operations on selected ones of the control data to effectuate maintenance of the control data, taking into account whether each burst transfer is a header burst transfer or a data burst transfer, if the burst transfer is a header burst transfer, whether the entire packet header has been completely transferred at the end of the particular burst transfer, and additionally, whether the packet is to be handled exceptionally.

20. The computer system as set forth in claim 18, wherein

the control data include for each channel a counter for tracking the length of the header remains to be burst transferred for the packet being burst transferred;
 for each burst transfer, the logic also uses the zero and non-zero states of the remaining header length counter to determine whether the burst transfer is a header burst transfer or a data burst transfer; and
 for each header burst transfer, the logic decrements the remaining header length counter by the block size, but not beyond zero.

21. The computer system as set forth in claim 20, wherein

the control data further include for each channel a normalized partial offset for identifying a location within a partially unloaded storage bucket of the channel where residual header/data starts, if the partially unloaded storage bucket exists; and

for each burst transfer, the logic computes an unnormalized partial offset by incrementing the normalized partial offset by either the block size or the pre-burst transfer value of the remaining header length counter, depending on whether the burst transfer is a header burst transfer or a data burst transfer, and if it is a header burst transfer, further depending on whether the remaining header length counter has been decremented to zero.

22. The computer system as set forth in claim 21, wherein,

the logic logically divides potential values that could be assumed by the unnormalized partial offset after each burst transfer into a plurality of ranges, each range denoting a portion of the linked storage buckets;

for each burst transfer, the logic further determines which portion of the linked storage buckets the unnormalized partial offset is pointing to by determining which range numerically encompasses the unnormalized partial offset; the control data further include a partial bucket pointer for identifying the partially unloaded storage bucket for the channel, a first bucket pointer for identifying the first full storage bucket for the channel, and a next bucket pointer for identifying the storage bucket immediately following the first storage bucket for the channel;

for each burst transfer, the logic further conditionally return the partially unloaded storage bucket, the first full storage bucket and the next storage bucket, and maintaining the partial, first and next bucket pointers, depending on which portion of the linked storage buckets the unnormalized partial offset is pointing to.

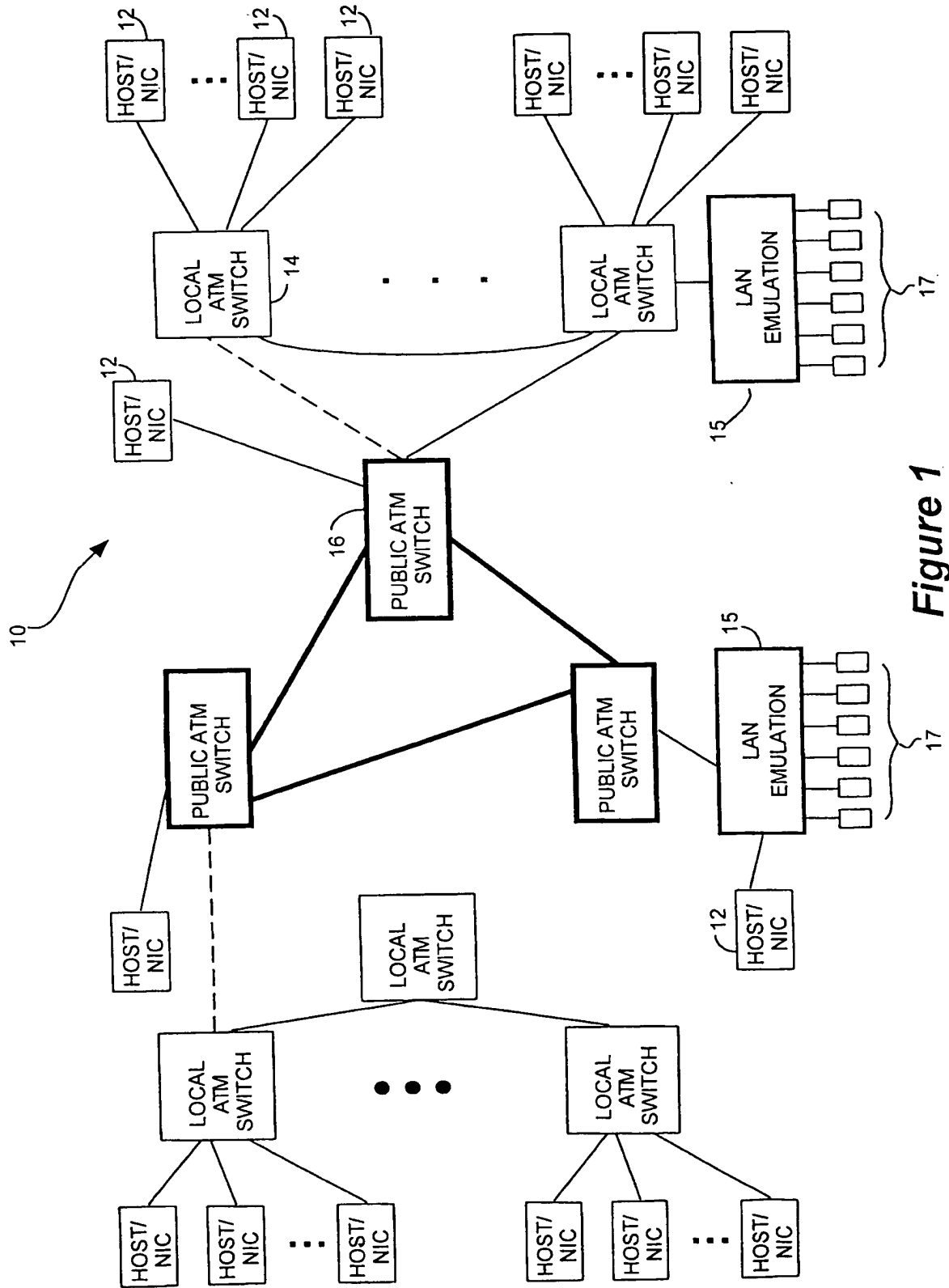


Figure 1

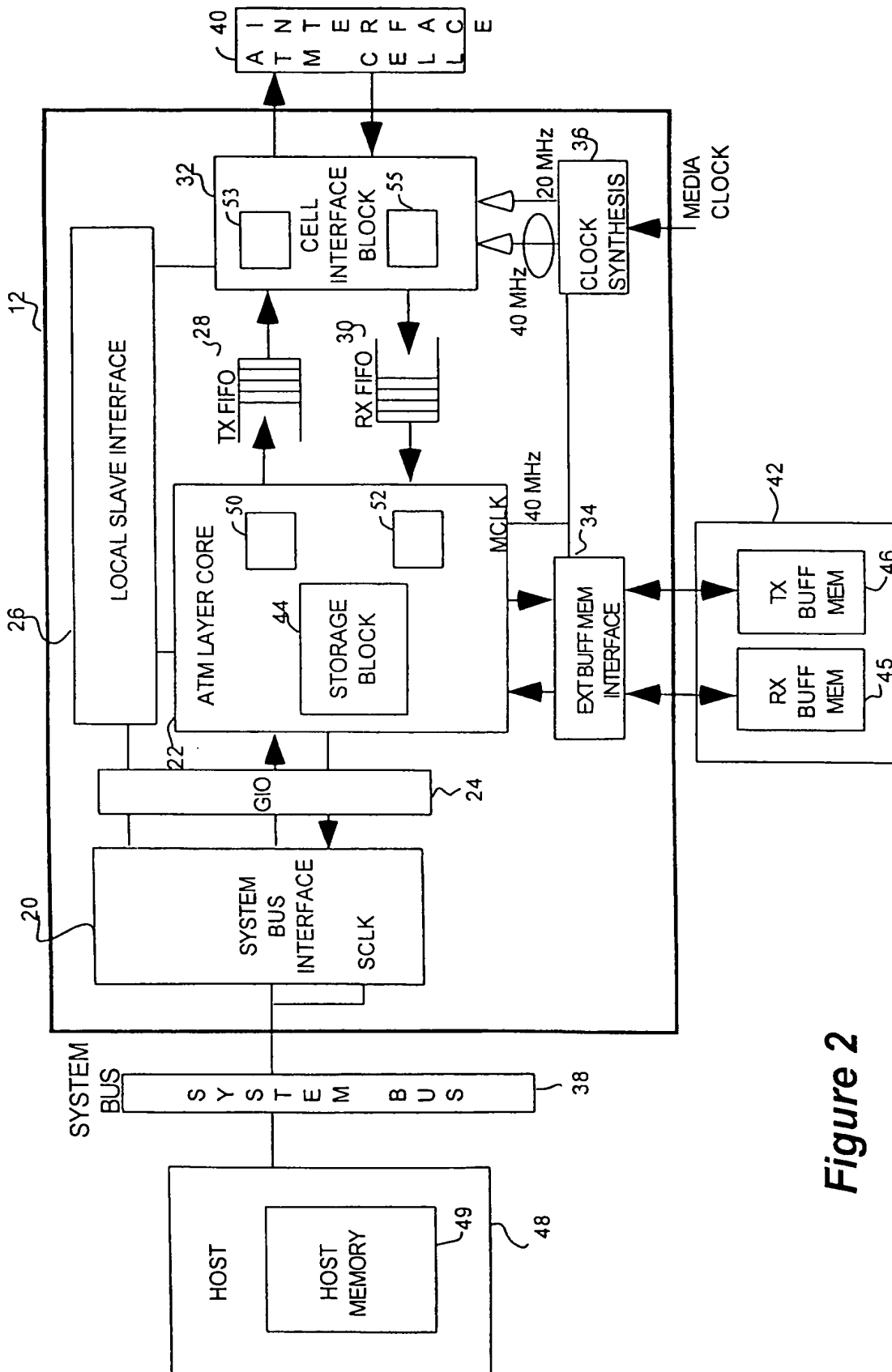
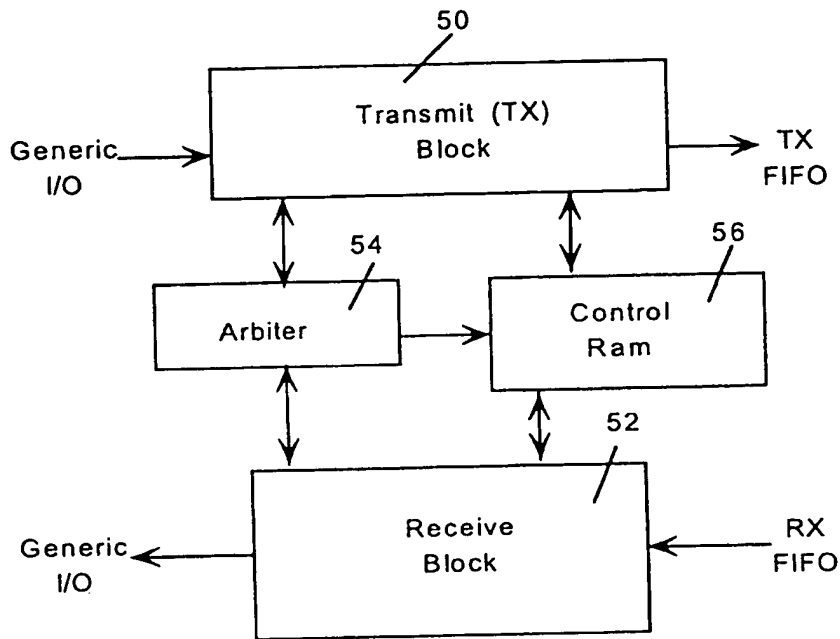
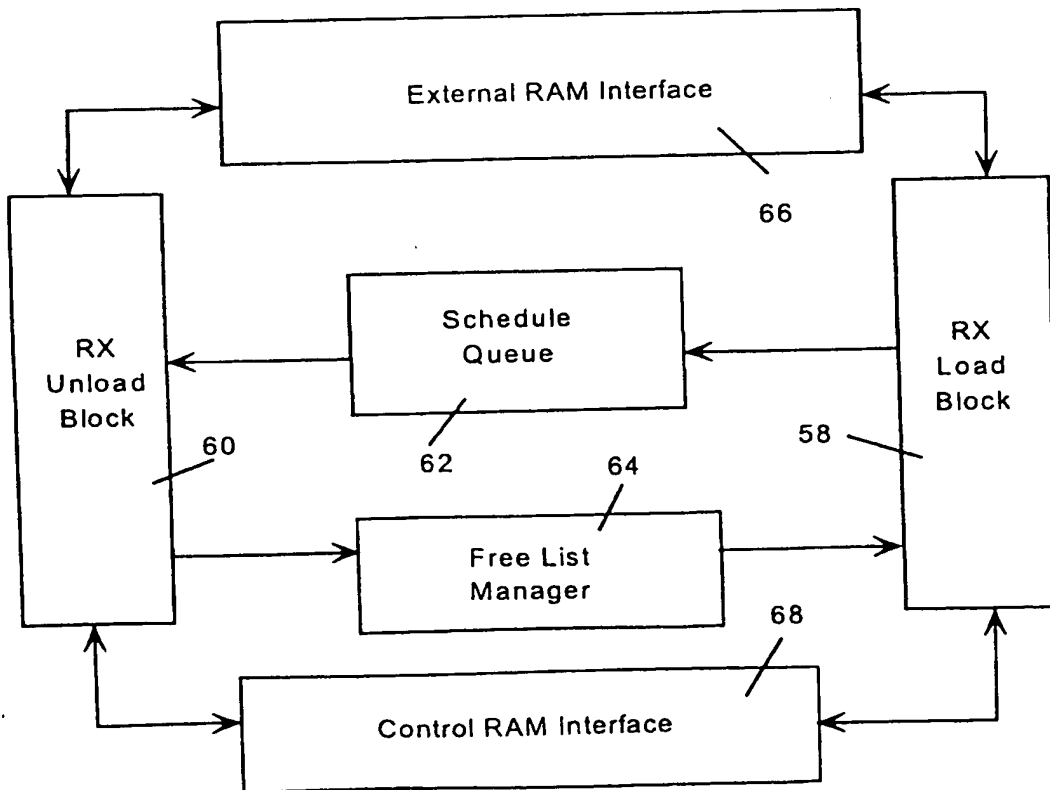


Figure 2



22

Figure 3



52

Figure 4

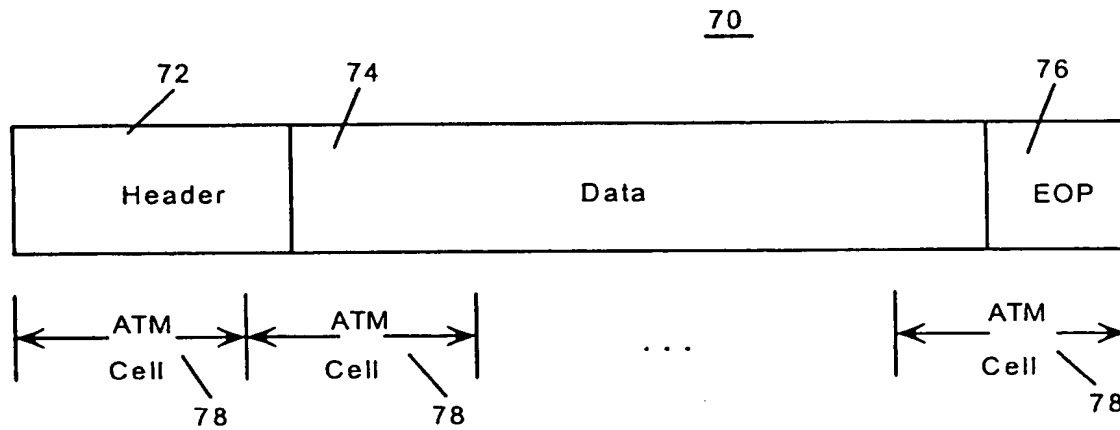


Figure 5

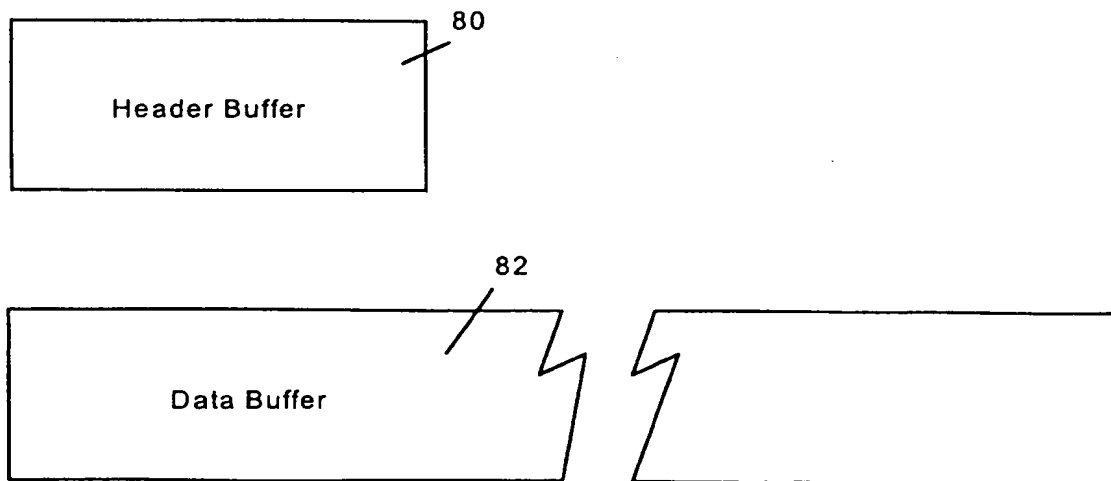


Figure 6

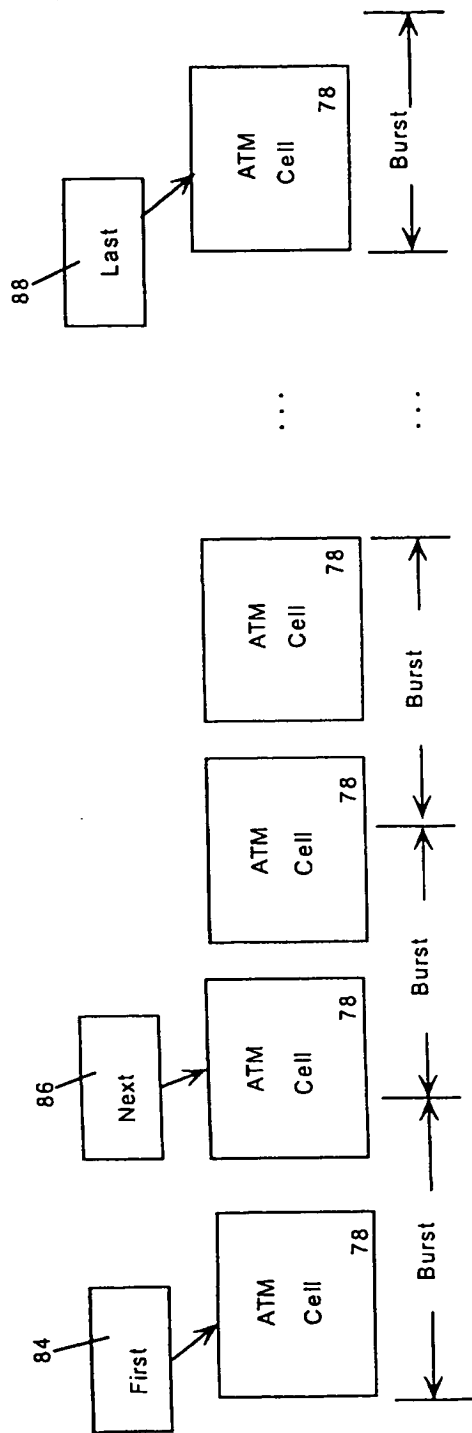


Figure 7

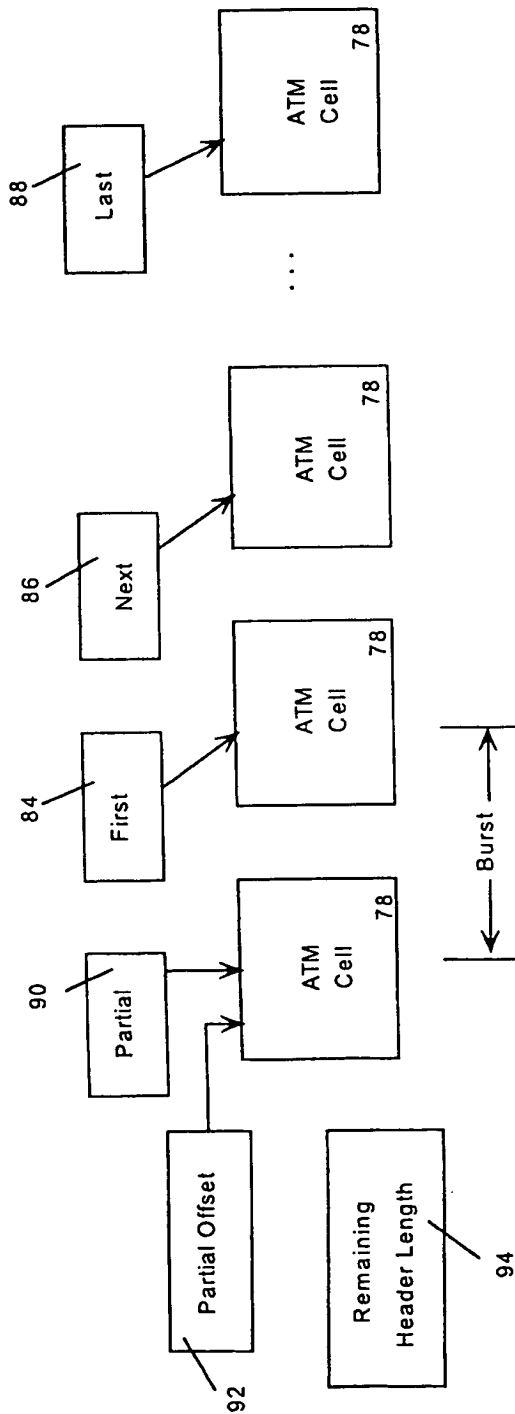


Figure 8

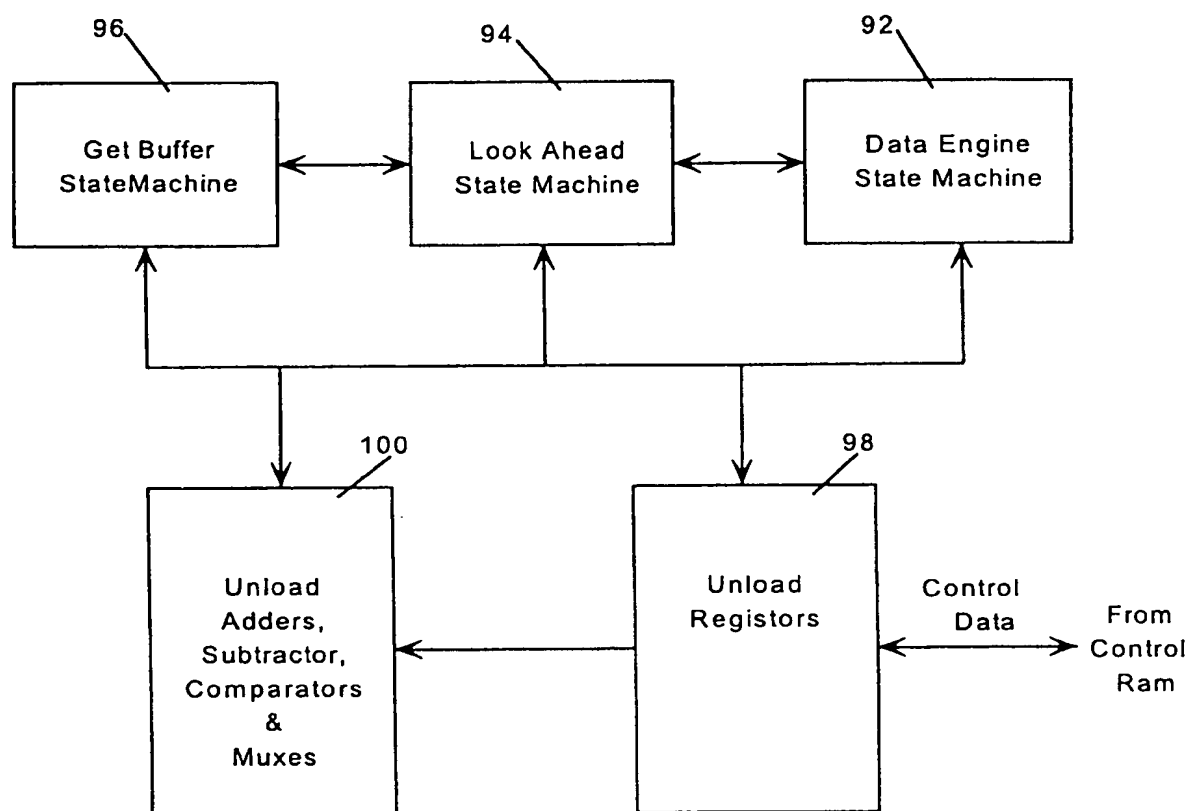


Figure 9

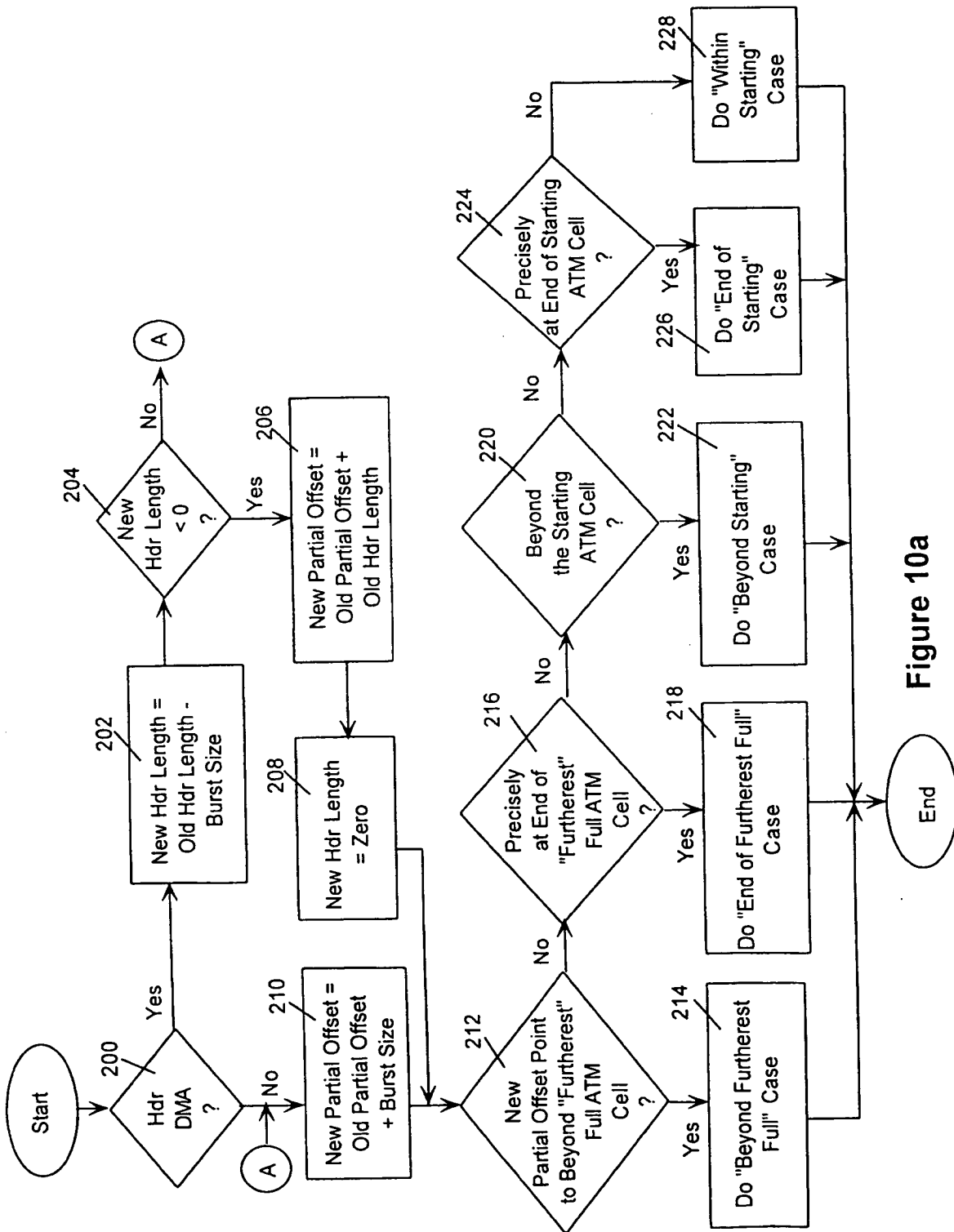


Figure 10a

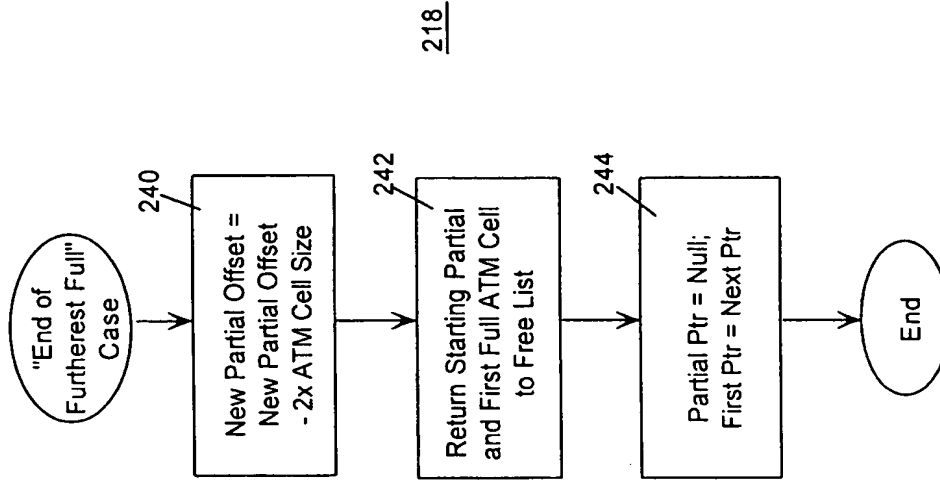


Figure 10c

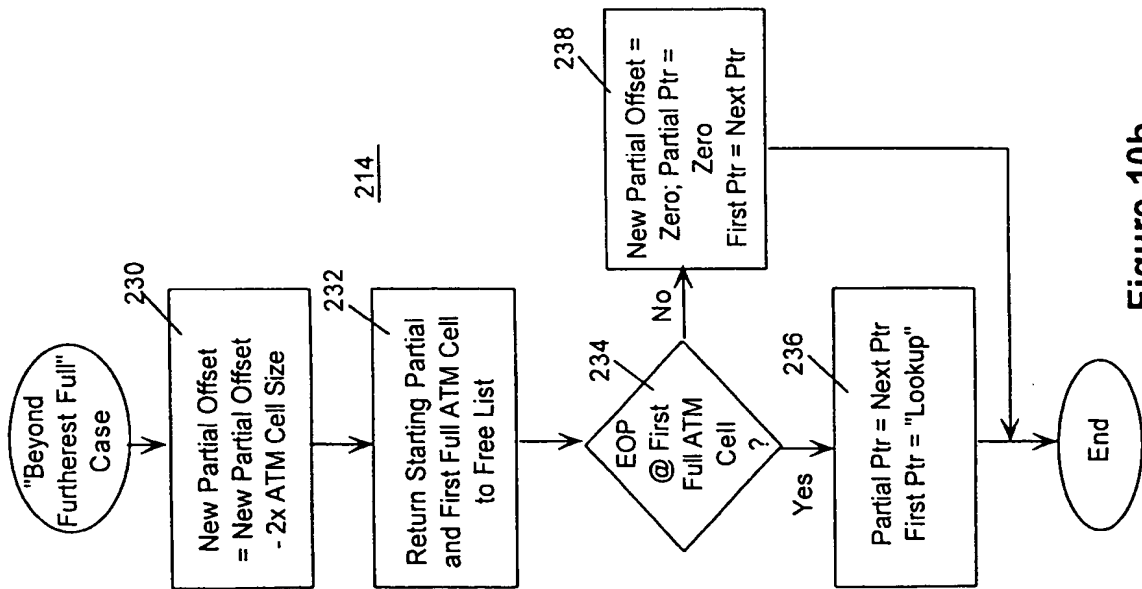


Figure 10b

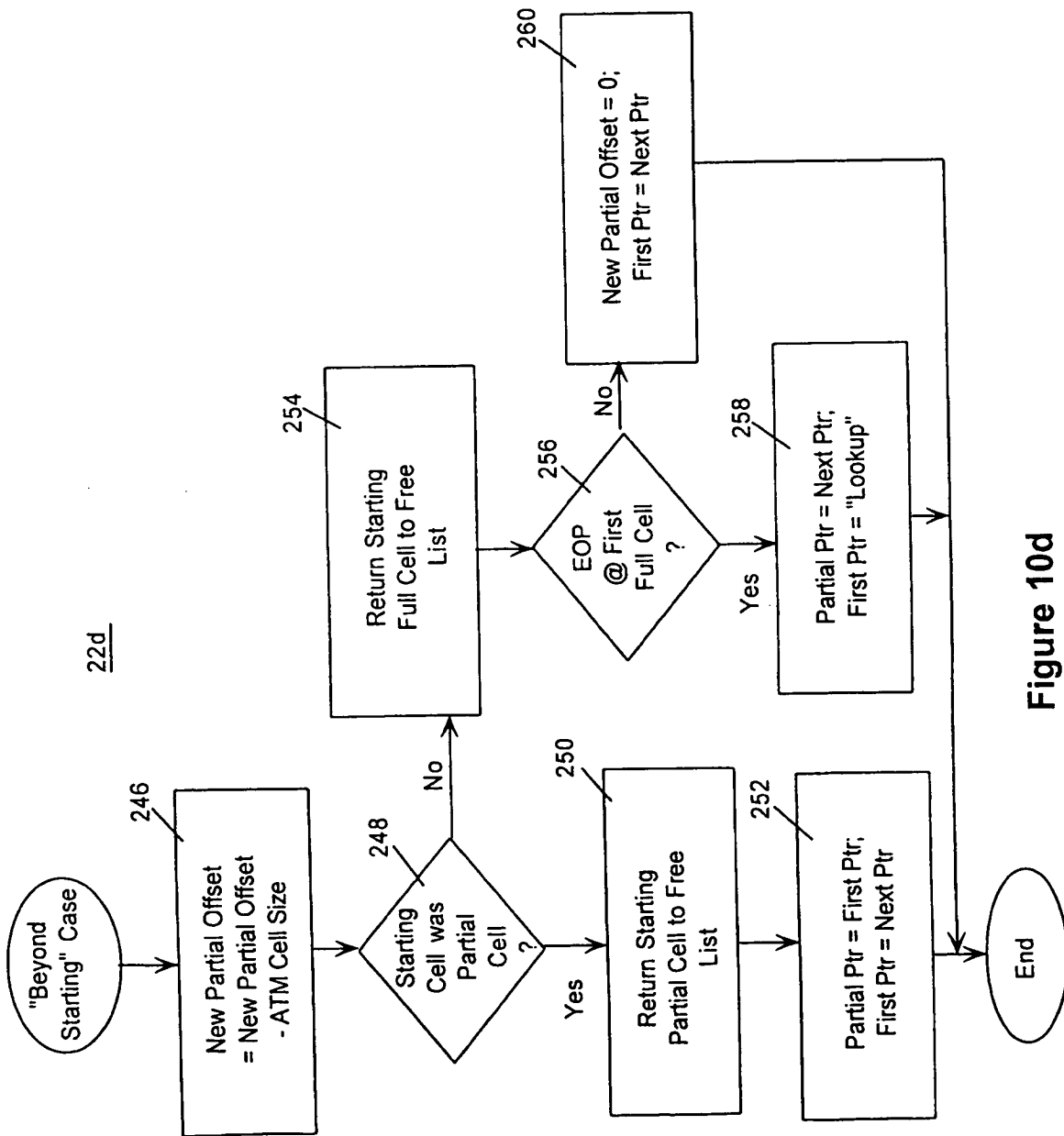


Figure 10d

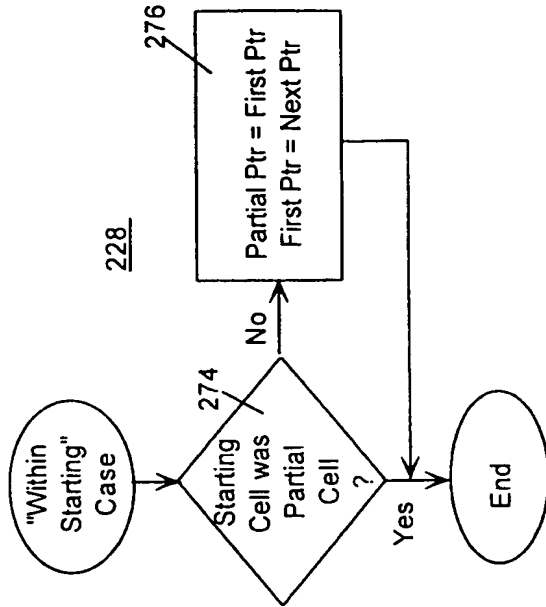


Figure 10f

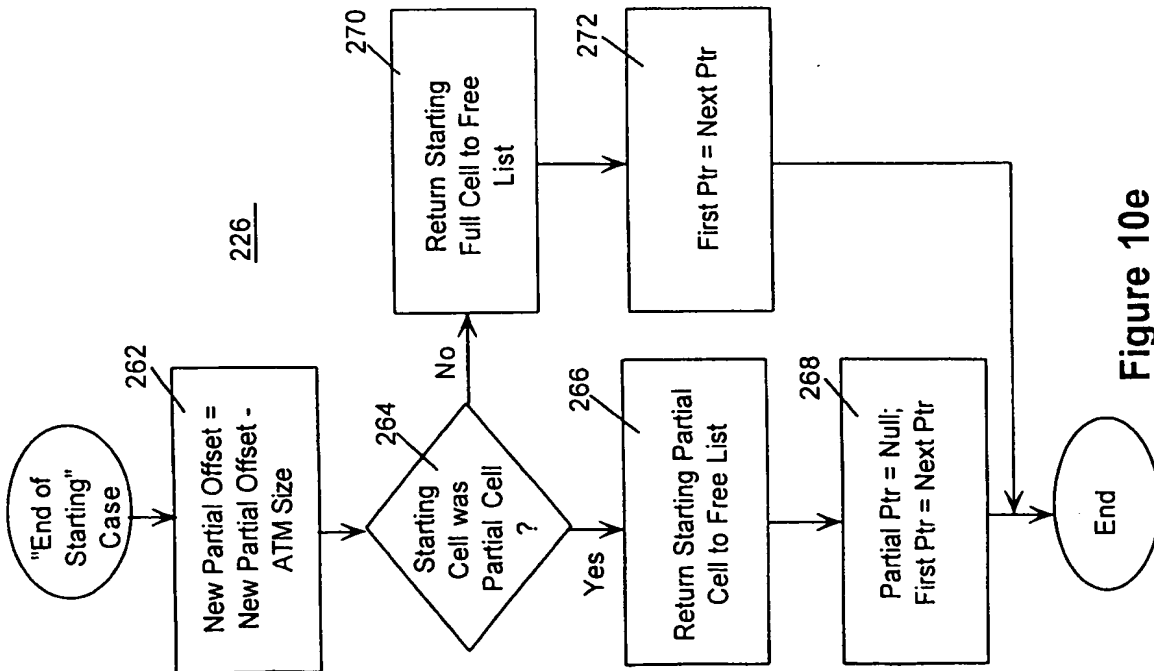
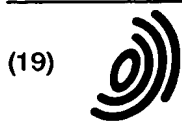


Figure 10e

This Page Blank (uspio,



Europäisches Patentamt

European Patent Office

Office européen des brevets



(11)

EP 0 772 368 A3

(12)

EUROPEAN PATENT APPLICATION

(88) Date of publication A3:
15.09.1999 Bulletin 1999/37

(51) Int. Cl.⁶: H04Q 11/04

(43) Date of publication A2:
07.05.1997 Bulletin 1997/19

(21) Application number: 96117003.2

(22) Date of filing: 23.10.1996

(84) Designated Contracting States:
DE FR GB NL SE

(30) Priority: 02.11.1995 US 552342

(71) Applicant:
SUN MICROSYSTEMS, INC.
Mountain View, CA 94043 (US)

(72) Inventors:
• Gentry, Denny E.
Palo Alto, California 94306 (US)

• Oskouy, Rasoul M.
Fremont, California 94539 (US)

(74) Representative:
Schmidt, Steffen J., Dipl.-Ing.
Wuesthoff & Wuesthoff,
Patent- und Rechtsanwälte,
Schweigerstrasse 2
81541 München (DE)

(54) Method and apparatus for burst transferring ATM packet header and data to a host computer system

(57) A network interface circuit (NIC) is provided with logic for maintaining various control pointers and at least one control counter for controlling burst transferring of buffered ATM cells to its host computer system in a non-cell-boundary-aligned block manner, distinguishing the ATM packet header from the ATM data most of the time, except for a number of predetermined exceptions. More specifically, ATM packet headers and ATM data are to be burst transferred to separate header and data buffers on the host computer system, except for short and atypical packets, in fixed size blocks, where the block size is complementary to the interface bus, but

not necessarily aligned with the ATM cell boundaries. For the short and atypical packets, both the header and data are to be burst transferred into the header buffer instead. The logic employs a two phase approach to determining the appropriate updates to the relevant control pointers and at least one control counter after each burst transfer of header/data to the header/data buffer. In one embodiment, the logic is provided to the lookahead state machine of an unload block, which is part of the receive block of a system and ATM layer core of the NIC.

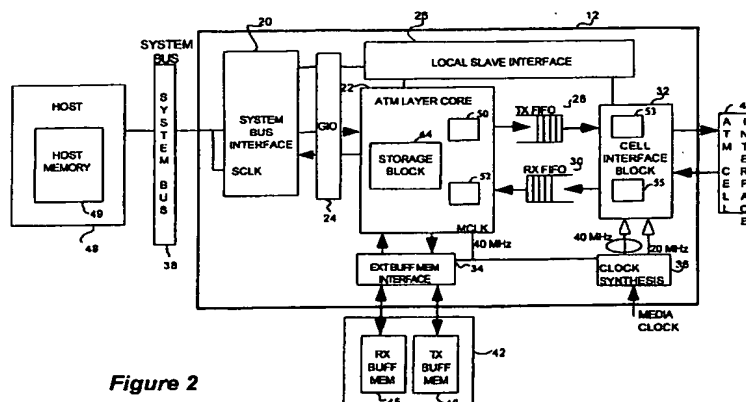


Figure 2

EP 0 772 368 A3



European Patent
Office

EUROPEAN SEARCH REPORT

Application Number
EP 96 11 7003

DOCUMENTS CONSIDERED TO BE RELEVANT			
Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
A	US 5 303 302 A (BURROWS MICHAEL) 12 April 1994 * abstract * * column 3, line 60 - column 5, line 53; figure 3 * * column 7, line 17 - column 8, line 45; figure 4 *	1,10,18	H04Q11/04
A	EP 0 674 461 A (SUN MICROSYSTEMS INC) 27 September 1995 * abstract * * column 2, line 53 - column 4, line 9 * * column 4, line 44 - column 5, line 11; figure 1 * * column 7, line 2 - column 7, line 56; figure 3 * * column 9, line 43 - column 10, line 50 *	1,10,18	
A	WO 95 11554 A (LSI LOGIC CORP) 27 April 1995 * abstract; figure 4 * * page 49, line 24 - page 50, line 8 * * page 54, line 2 - page 58, line 22 * * page 92, line 24 - page 93, line 24 *	1,10,18	TECHNICAL FIELDS SEARCHED (Int.Cl.6) H04Q
A	US 5 136 584 A (HEDLUND KURT A) 4 August 1992 * abstract * * column 7, line 17 - line 30; figure 4 *	1,10,18	
The present search report has been drawn up for all claims			
Place of search MUNICH		Date of completion of the search 16 July 1999	Examiner von der Straten, G
CATEGORY OF CITED DOCUMENTS X : particularly relevant if taken alone Y : particularly relevant if combined with another document of the same category A : technological background O : non-written disclosure P : intermediate document		T : theory or principle underlying the invention E : earlier patent document, but published on, or after the filing date D : document cited in the application L : document cited for other reasons & : member of the same patent family, corresponding document	

EPO FORM 1503 03 82 (P4/C01)

**ANNEX TO THE EUROPEAN SEARCH REPORT
ON EUROPEAN PATENT APPLICATION NO.**

EP 96 11 7003

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report.
The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

16-07-1999

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5303302 A	12-04-1994	NONE	
EP 0674461 A	27-09-1995	CA 2143495 A	22-09-1995
		JP 7297842 A	10-11-1995
		US 5859856 A	12-01-1999
WO 9511554 A	27-04-1995	US 5446726 A	29-08-1995
		AU 7981494 A	08-05-1995
		EP 0724794 A	07-08-1996
		JP 9504149 T	22-04-1997
		US 5856975 A	05-01-1999
		US 5708659 A	13-01-1998
		US 5864554 A	26-01-1999
		US 5914955 A	22-06-1999
		US 5872784 A	16-02-1999
		US 5654962 A	05-08-1997
		US 5887187 A	23-03-1999
US 5136584 A	04-08-1992	NONE	

EPO FORM P0459

For more details about this annex : see Official Journal of the European Patent Office, No. 12/82

This Page Blank (uspto)